2

Real-Time Scheduling and Resource Management

Giorgio C. Buttazzo
Scuola Superiore Sant'Anna

2.1 Introduction

A real-time control system is a system in which the resulting performance depends not only on the correctness of the single control actions but also on the time at which the actions are produced [30]. Real-time applications span a wide range of domains including industrial plants control, automotive, flight control systems, monitoring systems, multimedia systems, virtual reality, interactive games, consumer electronics, industrial automation, robotics, space missions, and telecommunications. In these systems, a late action might cause a wrong behavior (e.g., system instability) that could even lead to a critical system failure. Hence, 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, which is the maximum time allowed for a computational process to finish its execution.

The operating system is the major architectural component responsible for ensuring a timely execution of all the tasks having some timing requirements. In the presence of several concurrent activities running on a single processor, the objective of a real-time kernel is to ensure that each activity completes its execution within its deadline. Notice that this is very different than minimizing the average response times of a set of tasks.

Depending on the consequences caused by a missed deadline, real-time activities can be classified into hard and soft tasks. A real-time task is said to be hard if missing a deadline may have catastrophic consequences on the controlled system, and is said to be soft if missing a deadline causes a performance degradation but does not jeopardize the correct system behavior. An operating system able to manage hard tasks is called a hard real-time system [11,31]. 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. In general, hard real-time systems have to handle both hard and soft activities.

In spite of the large range of application domains, most of today's real-time control software is 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 practice is that the resulting control software 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.

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, direct memory access (DMA), cache, and prefetching mechanisms. Although such features improve the average performance of the processor, they introduce a nondeterministic 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.

2.1.1 Models and Terminology

To analyze the timing behavior of a real-time system, all software activities running in the processor are modeled as a set of n real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where each task τ_i is a sequence of instructions that, in the absence of other activities, is cyclicly executed on different input data. Hence, a task τ_i can be considered as an infinite sequence of instances, or jobs, $\tau_{i,j}$ $(j = 1, 2, \ldots)$, each having a computation time $c_{i,j}$, a release time $r_{i,j}$, and an absolute deadline $d_{i,j}$. For simplicity, all jobs of the same task $\tau_{i,j}$ are assumed to have the same worst-case execution time (WCET) C_i and the same relative deadline D_i, which is the interval of time, from the job release, within which the job should complete its execution.

In addition, the following timing parameters are typically defined on real-time tasks:

- $s_{i,j}$ denotes the start time of job $\tau_{i,j}$, that is, the time at which its first instruction is executed;
- $f_{i,j}$ denotes the finishing time of job $\tau_{i,j}$, that is, the time at which the job completes its execution;
- $R_{i,j}$ denotes the response time of job $\tau_{i,j}$, that is, the difference between the finishing time and the release time $(R_{i,j} = f_{i,j} - r_{i,j})$;
- R_i denotes the maximum response time of task τ_i, that is, $R_i = \max_j R_{i,j}$.

A task is said to be *periodic* if all its jobs are released one after the other with a regular interval T_i called the task period. If the first job $\tau_{i,1}$ is released at time $r_{i,1} = \Phi_i$ (also called the task phase), the generic job $\tau_{i,k}$ is characterized by the following release times and deadlines:

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

If the jobs are released in a nonregular fashion, the task is said to be *aperiodic*. Aperiodic tasks in which consecutive jobs are separated by a minimum interarrival time are called *sporadic*.

In real-time applications, timing constraints are usually specified on task execution, activation, or termination to enforce some performance requirements. In addition, other types of constraints can be defined on tasks, as precedence relations (for respecting some execution ordering) or synchronization points (for waiting for events or accessing mutually exclusive resources).

A schedule is said to be *feasible* if all tasks complete their execution under a set of specified constraints. A task set is said to be *schedulable* if there exists a feasible schedule for it.

Unfortunately, the problem of verifying the feasibility of a schedule in its general form has been proved to be NP-complete [15], and hence computationally intractable. However, the complexity of the feasibility analysis can be reduced for specific types of tasks and under proper (still significant) hypotheses.

In the rest of this chapter, a number of methods are presented to verify the schedulability of a task set under different assumptions. In particular, Section 2.2 treats the problem of scheduling and analyzing a set of periodic tasks; Section 2.3 addresses the issue of aperiodic service; Section 2.4 analyzes the effect of resource contention; Section 2.5 proposes some methods for handling overload conditions; and Section 2.6 concludes the chapter by presenting some open research problems.

2.2 Periodic Task Handling

Most of the control activities, such as signal acquisition, filtering, sensory data processing, action planning, and actuator control, are typically implemented as periodic tasks activated at specific rates imposed by the application requirements. When a set $\mathcal{T}$ of n periodic tasks has to be concurrently executed on the same processor, the problem is to verify whether all tasks can complete their execution within their timing constraints. In the rest of this section, we consider the analysis for a classical cyclic scheduling approach, a fixed priority-based scheduler, and a dynamic priority algorithm based on absolute deadlines.

2.2.1 Timeline Scheduling

One of the most commonly used approaches to schedule a set of periodic tasks on a single processor consists in dividing the timeline into slots of equal length and statically allocating tasks into slots to respect the constraints imposed by the application requirements. A timer synchronizes the activation of the tasks at the beginning of each slot. The length of the slot, called the minor cycle ($T_{\min}$), is set equal to the greatest common divisor of the periods, and the schedule has to be constructed until the least common multiple of all the periods, called the major cycle (T_{maj}) or the hyperperiod. Note that, since the schedule repeats itself every major cycle, the schedule has to be constructed only in the first N slots, where $N = T_{\text{maj}}/T_{\min}$. For such a reason, this method is also known as a *cyclic executive*.

To verify the feasibility of the schedule, it is sufficient to check whether the sum of the computation times in each slot is less than or equal to $T_{\min}$. If $h(i,j)$ is a binary function, equal to 1 if τ_i is allocated in slot j, and equal to 0 otherwise, the task set is schedulable if and only if

$$\forall j = 1 \ldots N, \quad \sum_{i=1}^{n} h(i,j)C_i < T_{\min}$$

To illustrate this method, consider the following example in which three tasks, τ_1, τ_2, and τ_3, with worst-case computation times $C_1 = 10$, $C_2 = 8$, and $C_3 = 5$, have to be periodically executed on a processor

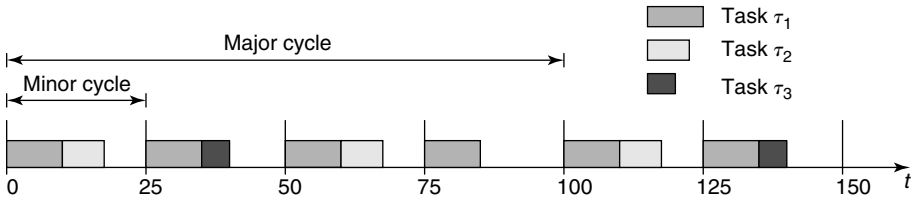

FIGURE 2.1 Example of timeline scheduling.

with periods $T_1 = 25$, $T_2 = 50$, and $T_3 = 100$ ms. From the task periods, the minor cycle results to be $T_{\min} = 25$ ms, whereas the major cycle results to be $T_{\text{maj}} = 100$ ms. To meet the required rates, it is easy to see that τ_1 needs to be executed in each slot ($T_1/T_{\min} = 1$), τ_2 every two slots ($T_2/T_{\min} = 2$), and τ_3 every four slots ($T_3/T_{\min} = 4$). A possible scheduling solution for this task set is illustrated in Figure 2.1.

Notice that the schedule produced by this method is feasible because

$$\begin{cases} C_1 + C_2 = 18 \leq 25 \text{ ms} \\ C_1 + C_3 = 15 \leq 25 \text{ ms} \end{cases}$$

The major relevant advantage of timeline scheduling is its simplicity. In fact, the method can be easily implemented by coding the major cycle into a big loop containing the task calls as they appear in the schedule. Then, a synchronization primitive has to be inserted before the task calls relative to each minor cycle and a timer must be programmed to generate an interrupt with a period equal to $T_{\min}$, so triggering the execution of the tasks allocated in each slot. Since the task sequence is not generated by a scheduling algorithm, but directly coded in a 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 and is not affected by jitter. In spite of these advantages, however, timeline scheduling also has some problems. For example, it is very fragile during overload conditions. In fact, if a task does not terminate at the minor cycle boundary and it is not aborted, it can cause a domino effect on the other tasks, breaking the entire schedule (timeline break). In contrast, if the failing task is aborted, it may leave a shared resource in an inconsistent state, so jeopardizing the correct system behavior.

Another big problem of the timeline scheduling technique is its sensitivity to application changes. If updating a task requires an increase in its computation time or its activation frequency, the entire scheduling sequence may need to be reconstructed from scratch. Considering the previous example, if task τ_2 is updated to τ_2' and the code is changed so that $C_1 + C_2' > 25$ ms, then task τ_2' must be divided into two or more pieces to be allocated in the available intervals in the timeline. Changing a task period may cause even more radical changes in the schedule. For example, if the period of task τ_2 changes from 50 to 40 ms, the previous schedule is not valid anymore, because the new minor cycle is equal to 10 ms and the new major cycle is equal to 200 ms, so a new schedule has to be constructed in $N = 20$ slots. 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.

2.2.2 Fixed-Priority Scheduling

The most common priority-based method for scheduling a set of periodic tasks is the rate-monotonic (RM) algorithm, which 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 fixed-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 nonpreemptive mode. In 1973, Liu and Layland [22] 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 $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ of n periodic tasks is schedulable by RM if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{2.1}$$

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. The right-hand term in Equation 2.1 decreases with n and, for large n, it tends to the following limit value:

$$\lim_{n \to \infty} n(2^{1/n} - 1) \;=\; \ln 2 \;\simeq\; 0.69 \tag{2.2}$$

The Liu and Layland test gives only a sufficient condition for the schedulability of a task set under the RM algorithm, meaning that, if Equation 2.1 is satisfied, then the task set is certainly schedulable, but if Equation 2.1 is not satisfied nothing can be said unless $U > 1$.

The schedulability of a task set under RM can also be checked using the hyperbolic test [6], according to which a task set is schedulable by RM if

$$\prod_{i=1}^{n} \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \tag{2.3}$$

Although only sufficient, the hyperbolic test is more precise than the Liu and Layland one in the sense that it is able to discover a higher number of schedulable task sets. Their difference can be better appreciated by representing the corresponding feasibility regions in the utilization space, denoted as the U-space. Here, the Liu and Layland bound (LL-bound) for RM is represented by an n-dimensional plane, which intersects each axis in $U_{\text{lub}}(n) = n(2^{1/n} - 1)$. All points below such a plane represent periodic task sets that are feasible by RM. The hyperbolic bound (H-bound) expressed by Equation 2.3 is represented by an n-dimensional hyperbolic surface tangent to the RM plane and intersecting the axes for $U_i = 1$. The hyperplane intersecting each axes in $U_i = 1$, denoted as the earliest deadline first (EDF)-bound, represents the limit of the feasibility region, above which any task set cannot be scheduled by any algorithm. Figure 2.2 illustrates such bounds for $n = 2$. From the plots, it is clear that the feasibility region below the H-bound is larger than that below the LL-bound, and the gain is given by the dark gray area. It is worth noting that such gain (in terms of schedulability) increases as a function of n and tends to $\sqrt{2}$ for n tending to infinity.

A necessary and sufficient schedulability test for RM is possible but at the cost of a higher computational complexity. Several pseudopolynomial time exact tests have been proposed in the real-time literature following different approaches [3,5,17,19]. For example, the method proposed by Audsley et al. [3], known as the response time analysis, consists in computing the worst-case response time R_i of each periodic task and then verifying that it does not exceed its relative deadline D_i. The worst-case response time of a task is derived by summing its computation time and the interference caused by tasks with higher priority:

$$R_i = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k \tag{2.4}$$

where $hp(i)$ denotes the set of tasks having priority higher than τ_i and $\lceil x \rceil$ the ceiling of a rational number, that is, the smallest 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, the iteration is stopped and the task set is declared unschedulable by RM.

All exact tests are more general than those based on the utilization because they also apply to tasks with relative deadlines less than or equal to periods. In this case, however, the scheduling algorithm that

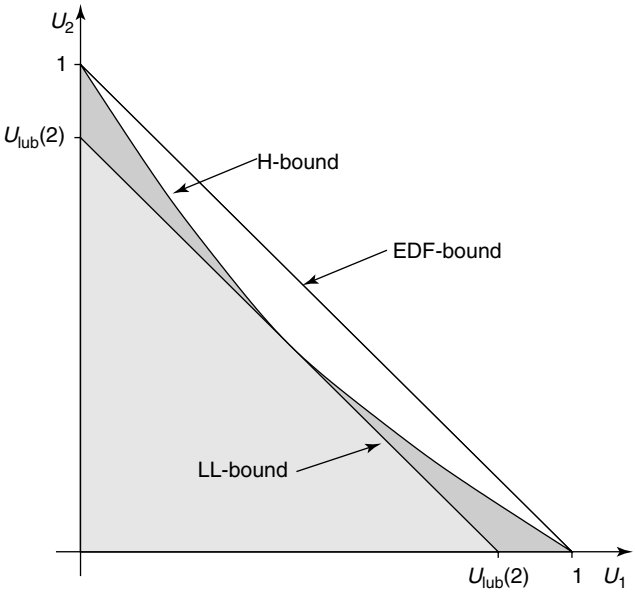

FIGURE 2.2 Schedulability bounds for RM and EDF in the utilization space.

achieves the best performance in terms of schedulability is the one that assigns priorities to tasks based on their relative deadlines, known as deadline monotonic (DM) [21]. 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 inversely proportional to its relative deadline. Since D_i is fixed for each task, DM is classified as a fixed-priority scheduling algorithm.

The major problem of fixed-priority scheduling is that, to achieve a feasible schedule, the processor cannot be fully utilized, except for the specific case in which the tasks have harmonic period relations (i.e., for any pair of tasks, one of the periods must be the multiple of the other). In the worst case, the maximum processor utilization that guarantees feasibility is about 0.69, as given by Equation 2.2. This problem can be overcome by dynamic priority scheduling schemes.

2.2.3 Earliest Deadline First

The most common dynamic priority scheme for real-time scheduling is the EDF algorithm, which orders the ready tasks based on their absolute deadline. According to EDF, a task receives the highest priority if its deadline is the earliest among those of the ready tasks. Since the absolute deadline changes from job to job in the same task, EDF is considered a dynamic priority algorithm. The EDF algorithm is typically preemptive in the sense that a newly arrived task preempts the running task if its absolute deadline is shorter. However, it can also be used in a nonpreemptive fashion.

EDF 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. In 1973, Liu and Layland [22] proved that a set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ of n periodic tasks is schedulable by EDF if and only if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.5}$$

Later, Dertouzos [14] showed that EDF is optimal among all online algorithms, meaning that if a task set is not schedulable by EDF, then it cannot be scheduled by any other algorithm. Note that Equation 2.5

provides a necessary and sufficient condition to verify the feasibility of the schedule. Thus, if it is not satisfied, no algorithm can produce a feasible schedule for that task set.

The dynamic priority assignment allows EDF to exploit the full CPU capacity, reaching up to 100% of processor utilization. When the task set has a utilization factor less than one, the residual fraction can be efficiently used to handle aperiodic requests activated by external events. In general, compared with fixed-priority schemes, EDF is superior in many aspects [7], and also generates a lower number of context switches, thus causing less runtime overhead. Finally, using a suitable kernel mechanism for time representation [13], EDF can be effectively implemented even in small microprocessors [8] for increasing system utilization and achieving a timely execution of periodic and aperiodic tasks.

Under EDF, the schedulability analysis for periodic task sets with deadlines less than periods is based on the processor demand criterion [4]. According to this method, a task set is schedulable by EDF if and only if, in every interval of length L, the overall computational demand is no greater than the available processing time, that is, if and only if $U < 1$ and

$$\forall L > 0, \quad \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i \leq L \tag{2.6}$$

where $\lfloor x \rfloor$ denotes the floor of a rational number, that is, the highest integer less than or equal to x. Notice that, in practice, the number of points in which the test has to be performed can be limited to the set of absolute deadlines not exceeding $t_{\max} = \min\{L^*, H\}$, where H is the hyperperiod and

$$L^* = \max \left\{ D_1, \ldots, D_n, \frac{\sum_{i=1}^{n} (T_i - D_i) C_i / T_i}{1 - U} \right\} \tag{2.7}$$

2.3 Handling Aperiodic Tasks

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 signaled through interrupts), which may arrive at irregular intervals of time. 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, 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 offline. 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 2.3, if the aperiodic request is serviced immediately (i.e., with a priority higher than that assigned to periodic tasks), then task τ_2 will miss its deadline.

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 timing constraints. Considering the same task set as before, Figure 2.4 illustrates how job J_a is handled by a background service.

The response time of aperiodic tasks can be improved by handling them through an aperiodic server dedicated to their execution. As any other periodic task, a server is characterized by a period T_s and

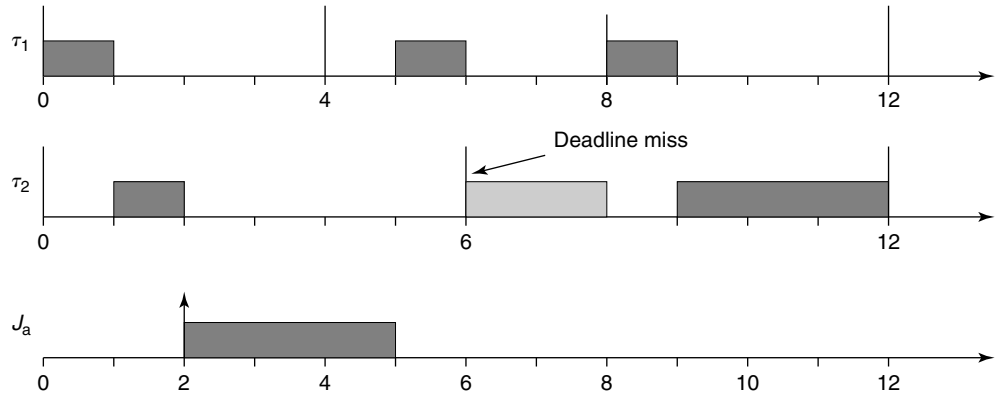

FIGURE 2.3 Immediate service of an aperiodic task. Periodic tasks are scheduled by RM.

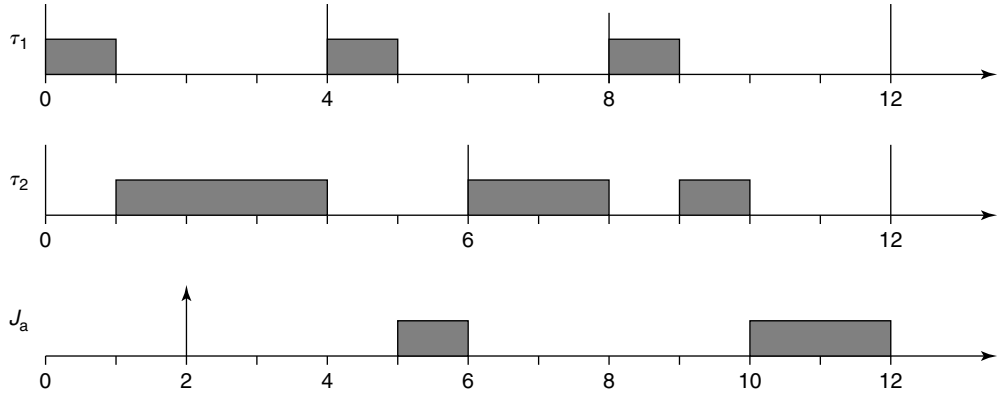

FIGURE 2.4 Background service of an aperiodic task. Periodic tasks are scheduled by RM.

an execution time C_s, called the server capacity (or budget). In general, the server is scheduled using the algorithm adopted for the 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 past 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 [20,32], the sporadic server [27], and the slack stealer [18]. Among those servers using dynamic priorities (which are more efficient on the average) we recall the dynamic sporadic server [16,28], the total bandwidth server [29], the tunable bandwidth server [10], and the constant bandwidth server (CBS) [1]. To clarify the idea behind an aperiodic server, Figure 2.5 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, the priority is given to the server to enhance aperiodic responsiveness. We also observe that the same task set would not be schedulable under a fixed-priority system.

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 that still produces a feasible EDF schedule [10]. The schedule generated by the optimal TB* algorithm is illustrated in Figure 2.6, 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, better performance is achieved at the price of a larger

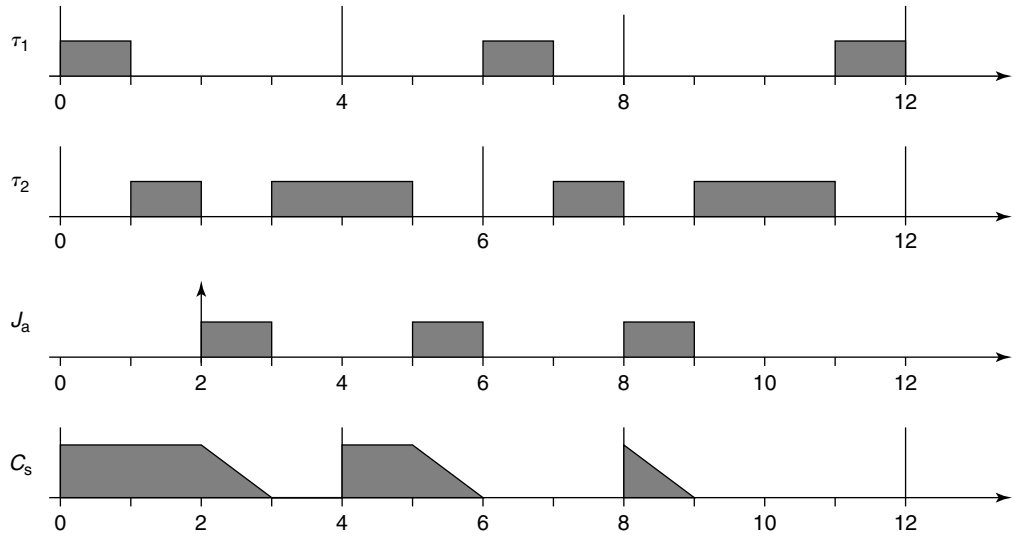

FIGURE 2.5 Aperiodic service performed by a dynamic deferrable server. Periodic tasks, including the server, are scheduled by EDF. C_s is the remaining budget available for J_a.

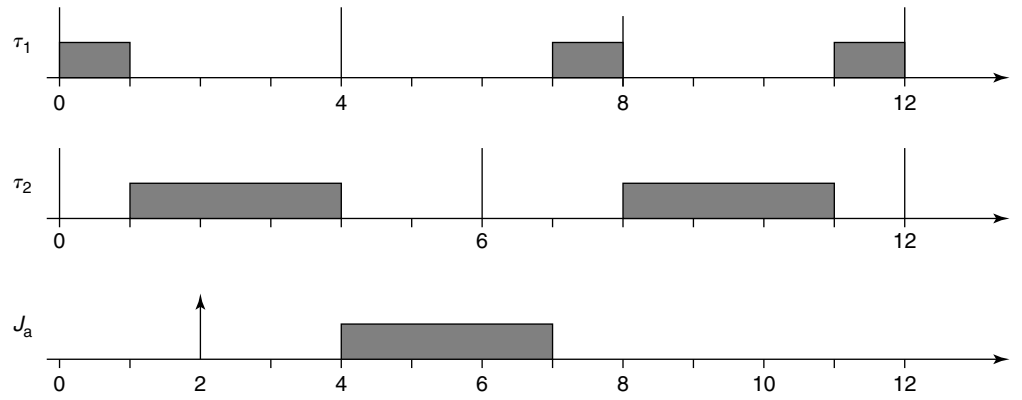

FIGURE 2.6 Optimal aperiodic service under EDF.

runtime overhead (due to the complexity of computing the minimum deadline). However, adopting a variant of the algorithm, called the tunable bandwidth server [10], overhead cost and performance can be balanced to select the best service method for a given real-time system. An overview of the most common aperiodic service algorithms (both under fixed and dynamic priorities) can be found in Ref. 11.

2.4 Handling 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

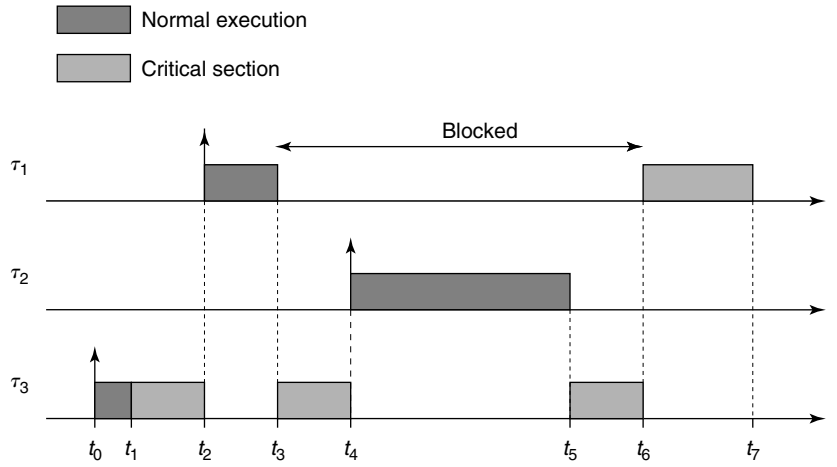

FIGURE 2.7 Example of priority inversion.

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 2.7, suppose that at time t_1 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.

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 (such as τ_2), which will prolong the blocking interval of τ_1 for their entire execution! The situation illustrated in Figure 2.7 can be avoided by simply preventing 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. For example, a low-priority task inside a long critical section could prevent the execution of high-priority tasks even though they do not share any resource. A more efficient solution is to regulate the access to shared resources through the use of specific concurrency control protocols [24] designed to limit the priority inversion phenomenon.

2.4.1 Priority Inheritance Protocol

An elegant solution to the priority inversion phenomenon caused by mutual exclusion is offered by the Priority Inheritance Protocol [26]. 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 2.8 illustrates how the schedule shown in Figure 2.7 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 2.7. 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).

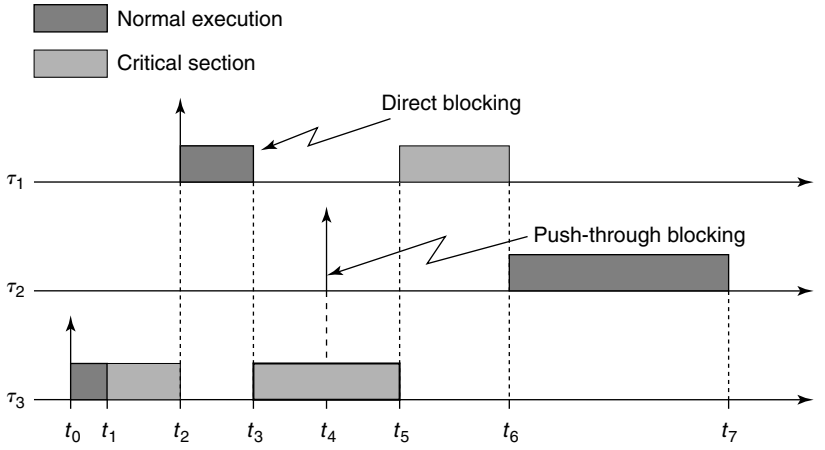

FIGURE 2.8 Schedule produced using priority inheritance on the task set of Figure 2.7.

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.

The Priority Inheritance Protocol has the following property [26]:

> Given a task τ_i, let l_i be the number of tasks with lower priority sharing a resource with a task with priority higher than or equal to τ_i, and let r_i be the number of resources that could block τ_i. Then, τ_i can be blocked for at most the duration of min (l_i, r_i) 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.

2.4.2 Priority Ceiling Protocol

The Priority Ceiling Protocol [26] 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.

2.4.3 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 nonpreemptable resource, for example, 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

$$\forall i, \quad 1 \leq i \leq n, \qquad \sum_{k \in hp(i)} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1) \tag{2.8}$$

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.

2.5 Overload Management

This section deals with the problem of scheduling real-time tasks in overload conditions; that is, in those critical situations in which the computational demand requested by the task set exceeds the time available on the processor, and hence not all tasks can complete within their deadlines.

A transient overload condition can occur for the simultaneous arrival of asynchronous events, or because some execution time exceeds the value for which it has been guaranteed. When a task executes more than expected, it is said to overrun. In a periodic task system, the overload can become permanent after the activation of a new periodic task (if $U > 1$), or after a task increases its activation rate to react to some change in the environment. In such a situation, computational activities start to accumulate in the system's queues (which tend to become longer and longer if the overload persists), and tasks' response times tend to increase indefinitely. In the following sections, we present two effective methods for dealing with transient and permanent overload conditions.

2.5.1 Resource Reservation

Resource reservation is a general technique used in real-time systems for limiting the effects of overruns in tasks with variable computation times. According to this method, each task is assigned a fraction of the available resources just enough to satisfy its timing constraints. The kernel, however, must prevent each task to consume more than the allocated amount to protect the other tasks in the systems (temporal protection). In this way, a task receiving a fraction U_i of the total processor bandwidth behaves as it were executing alone on a slower processor with a speed equal to U_i times the full speed. The advantage of this method is that each task can be guaranteed in isolation independent of the behavior of the other tasks.

A simple and effective mechanism for implementing temporal protection in a real-time system is to reserve, for each task τ_i, a specified amount Q_i of CPU time in every interval P_i. Some authors [25] tend to distinguish between *hard* and *soft* reservations, where a hard reservation allows the reserved task to execute *at most* for Q_i units of time every P_i, whereas a soft reservation guarantees that the task executes *at least* for Q_i time units every P_i, allowing it to execute more if there is some idle time available.

A resource reservation technique for fixed-priority scheduling was first presented in Ref. 23. According to this method, a task τ_i is first assigned a pair (Q_i, P_i) (denoted as a CPU *capacity reserve*) and then it is enabled to execute as a real-time task for Q_i units of time every interval of length P_i. When the task consumes its reserved quantum Q_i, it is blocked until the next period, if the reservation is hard, or it is scheduled in background as a nonreal-time task, if the reservation is soft. If the task is not finished, it is assigned another time quantum Q_i at the beginning of the next period and it is scheduled as a real-time task until the budget expires, and so on. In this way, a task is *reshaped* so that it behaves like a periodic real-time task with known parameters (Q_i, P_i) and can be properly scheduled by a classical real-time scheduler.

Under EDF, temporal protection can be efficiently implemented by handling each task through a dedicated CBS [1,2]. The behavior of the server is tuned by two parameters (Q_i, P_i), where Q_i is the *server maximum budget* and P_i is the *server period*. The ratio $U_i = Q_i/P_i$ is denoted as the *server bandwidth*. At each instant, two state variables are maintained for each server: the server deadline d_i and the actual server budget q_i. Each job handled by a server is scheduled using the current server deadline and whenever the

server executes a job, the budget q_i is decreased by the same amount. At the beginning $d_i = q_i = 0$. Since a job is not activated while the previous one is active, the CBS algorithm can be formally defined as follows:

1. When a job $\tau_{i,j}$ arrives, if $q_i \geq (d_i - r_{i,j})U_i$, it is assigned a server deadline $d_i = r_{i,j} + P_i$ and q_i is recharged at the maximum value Q_i, otherwise the job is served with the current deadline using the current budget.
2. When $q_i = 0$, the server budget is recharged at the maximum value Q_i and the server deadline is postponed at $d_i = d_i + P_i$. Notice that there are no finite intervals of time in which the budget is equal to zero.

As shown in Ref. 2, if a task τ_i is handled by a CBS with bandwidth U_i it will never demand more than U_i independent of the actual execution time of its jobs. As a consequence, possible overruns occurring in the served task do not create extra interference in the other tasks, but only delay τ_i.

Although such a method is essential for achieving predictability in the presence of tasks with variable execution times, the overall system performance becomes quite dependent on a correct bandwidth allocation. In fact, 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. In contrast, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

2.5.2 Period Adaptation

If a permanent overload occurs in a periodic task set, the load can be reduced by enlarging task periods to suitable values, so that the total workload can be kept below a desired threshold. The possibility of varying tasks' rates increases the flexibility of the system in handling overload conditions, providing a more general admission control mechanism. For example, whenever a new task cannot be guaranteed by the system, instead of rejecting the task, the system can try to reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request.

An effective method to change task periods as a function of the desired workload is the elastic framework [9,12], according to which each task is considered as flexible as a spring, whose utilization can be modified by changing its period within a specified range. The advantage of the elastic model with respect to the other methods proposed in the literature is that a new period configuration can easily be determined online as a function of the elastic coefficients, which can be set to reflect tasks' importance. Once elastic coefficients are defined based on some design criterion, periods can be quickly computed online depending on the current workload and the desired load level.

More specifically, each task is characterized by four parameters: a worst-case computation time C_i, a minimum period $T_{i_{\min}}$ (considered as a nominal period), a maximum period $T_{i_{\max}}$, and an elastic coefficient E_i. The elastic coefficient specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration: the greater the E_i, the more elastic the task. Hence, an elastic task is denoted by

$$\tau_i(C_i, T_{i_{\min}}, T_{i_{\max}}, E_i)$$

The actual period of task τ_i is denoted by T_i and is constrained to be in the range $[T_{i_{\min}}, T_{i_{\max}}]$. Moreover, $U_{i_{\max}} = C_i/T_{i_{\min}}$ and $U_{i_{\min}} = C_i/T_{i_{\max}}$ denote the maximum and minimum utilization of τ_i, whereas $U_{\max} = \sum_{i=1}^{n} U_{i_{\max}}$ and $U_{\min} = \sum_{i=1}^{n} U_{i_{\min}}$ denote the maximum and minimum utilization of the task set.

Assuming tasks are scheduled by the EDF algorithm [22], if $U_{\max} \leq 1$, all tasks can be activated at their minimum period $T_{i_{\min}}$, otherwise the elastic algorithm is used to adapt their periods to T_i such that $\sum \frac{C_i}{T_i} = U_{\mathrm{d}} \leq 1$, where U_{d} is some desired utilization factor. This can be done as in a linear spring system, where springs are compressed by a force F (depending on their elasticity) up to a desired total length. The concept is illustrated in Figure 2.9. It can be easily shown (see Ref. 9 for details) that a solution always exists if $U_{\min} \leq U_{\mathrm{d}}$.

events have to be provided within precise timing constraints to guarantee a desired level of performance. The combination of real-time features in tasks with dynamic behavior, together with cost and resource constraints, creates new problems to be addressed in the design of such systems at different architecture levels. The classical worst-case design approach, typically adopted in hard real-time systems to guarantee timely responses in all possible scenarios, is no longer acceptable in highly dynamic environments, because it would waste resources and prohibitively increase the cost. Instead of allocating resources for the worst case, smarter techniques are needed to sense the current state of the environment and react as a consequence. This means that, to cope with dynamic environments, a real-time system must be *adaptive*, that is, it must be able to adjust its internal strategies in response to a change in the environment to keep the system performance at a desired level or, if this is not possible, degrade it in a controlled fashion.

References

1. L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.
2. L. Abeni and G. Buttazzo, "Resource Reservation in Dynamic Real-Time Systems," *Real-Time Systems*, Vol. 27, No. 2, pp. 123–167, July 2004.
3. N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying New Scheduling Theory to Static Priority Preemptive Scheduling," *Software Engineering Journal*, Vol. 8, No. 5, pp. 284–292, September 1993.
4. 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*, Vol. 2, No. 4, pp. 301–304, 1990.
5. E. Bini and G. Buttazzo, "Schedulability Analysis of Periodic Fixed Priority Systems," *IEEE Transactions on Computers*, Vol. 53, No. 11, pp. 1462–1473, November 2004.
6. E. Bini, G. C. Buttazzo, and G. M. Buttazzo, "Rate Monotonic Analysis: The Hyperbolic Bound," *IEEE Transactions on Computers*, Vol. 52, No. 7, pp. 933–942, July 2003.
7. G. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day," *Real-Time Systems*, Vol. 28, pp. 1–22, 2005.
8. G. Buttazzo and P. Gai, "Efficient EDF Implementation for Small Embedded Systems," *Proceedings of the 2nd Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2006)*, Dresden, Germany, July 2006.
9. G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic Scheduling for Flexible Workload Management," *IEEE Transactions on Computers*, Vol. 51, No. 3, pp. 289–302, March 2002.
10. G. Buttazzo and F. Sensini, "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments," *IEEE Transactions on Computers*, Vol. 48, No. 10, pp. 1035–1052, October 1999.
11. G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Boston, MA, 1997.
12. G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control," *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 286–295, December 1998.
13. A. Carlini and G. Buttazzo, "An Efficient Time Representation for Real-Time Embedded Systems," *Proceedings of the ACM Symposium on Applied Computing (SAC 2003)*, Melbourne, FL, pp. 705–712, March 9–12, 2003.
14. M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Information Processing*, Vol. 74, North-Holland Publishing Company, pp. 807–813, 1974.
15. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.
16. T. M. Ghazalie and T. P. Baker, "Aperiodic Servers in a Deadline Scheduling Environment," *Real-Time Systems*, Vol. 9, No. 1, pp. 31–67, 1995.

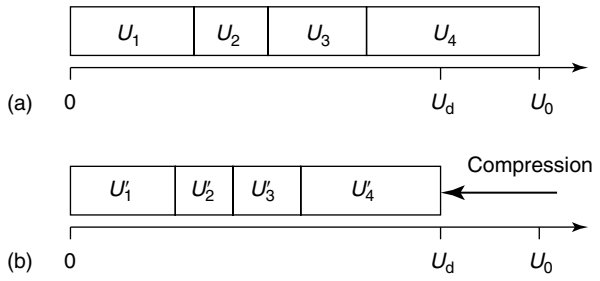

FIGURE 2.9 Compressing the utilizations of a set of elastic tasks.

As shown in Ref. 9, in the absence of period constraints (i.e., if $T_{\max} = \infty$), the utilization U_i of each compressed task can be computed as follows:

$$\forall i, \qquad U_i = U_{i_{\max}} - (U_{\max} - U_{\mathrm{d}}) \frac{E_i}{E_{\mathrm{tot}}} \tag{2.9}$$

where

$$E_{\mathrm{tot}} = \sum_{i=1}^{n} E_i \tag{2.10}$$

In the presence of period constraints, the compression algorithm becomes iterative with complexity $O(n^2)$, where n is the number of tasks. The same algorithm can be used to reduce the periods when the overload is over, so adapting task rates to the current load condition to better exploit the computational resources.

2.6 Conclusions

This chapter surveyed some kernel methodologies aimed at enhancing the efficiency and the predictability of real-time control applications. In particular, some scheduling algorithms and analysis techniques have been presented for periodic and aperiodic task sets illustrating that dynamic priority scheduling schemes achieve better resource exploitation with respect to fixed-priority algorithms.

It has been shown that, when tasks interact through shared resources, the use of critical sections may cause a priority inversion phenomenon, where high-priority tasks can be blocked by low-priority tasks for an unbounded interval of time. Two concurrency control protocols (namely the Priority Inheritance and Priority Ceiling protocols) have been described to avoid this problem. Each method allows to bound the maximum blocking time for each task and can be analyzed offline to verify the feasibility of the schedule within the timing constraints imposed by the application.

Finally, some overload management techniques have been described to keep the system workload below a desired threshold and deal with dangerous peak load situations that could degrade system performance. In the presence of soft real-time activities with extremely variable computation requirements (as those running in multimedia systems), *resource reservation* is an effective methodology for limiting the effects of execution overruns and protecting the critical activities from an unbounded interference. Moreover, it allows to guarantee a task in isolation independent of the behavior of the other tasks. Implementing resource reservation, however, requires a specific support from the kernel, which has to provide a tracing mechanism to monitor the actual execution of each job. To prevent permanent overload conditions caused by excessive periodic load, elastic scheduling provides a simple and effective way to shrink task utilizations up to a desired load.

Next generation embedded systems are required to work in dynamic environments where the characteristics of the computational load cannot always be predicted in advance. Still timely responses to

17. M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, Vol. 29, No. 5, pp. 390–395, 1986.
18. J. P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.
19. 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.
20. 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.
21. J. Leung and J. Whitehead, "On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks," *Perfomance Evaluation*, Vol. 2, No. 4, pp. 237–250, 1982.
22. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 40–61, 1973.
23. C. W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves for Multimedia Operating Systems," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, May 1994.
24. R. Rajkumar, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Boston, MA, 1991.
25. R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems," *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, San José, CA, January 1998.
26. L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175–1185, 1990.
27. B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time System," *Journal of Real-Time Systems*, Vol. 1, pp. 27–60, June 1989.
28. M. Spuri and G. C. Buttazzo, "Efficient Aperiodic Service under Earliest Deadline Scheduling," *Proceedings of IEEE Real-Time System Symposium*, San Juan, Puerto Rico, December 1994.
29. M. Spuri and G. C. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real-Time Systems*, Vol. 10, No. 2, pp. 179–210, 1996.
30. J. Stankovic, "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems", *IEEE Computer*, Vol. 21, No. 10, pp. 10–19, October 1988.
31. J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, Vol. 28, No. 6, pp. 16–25, June 1995.
32. 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*, Vol. 44, No. 1, pp. 73–91, January 1995.