6

DYNAMIC PRIORITY SERVERS

6.1   INTRODUCTION

In this chapter[1] we discuss the problem of scheduling soft aperiodic tasks and hard periodic tasks under dynamic priority assignments. In particular, different service methods are introduced, the objective of which is to reduce the average response time of aperiodic requests without compromising the schedulability of hard periodic tasks. Periodic tasks are scheduled by the Earliest Deadline First (EDF) algorithm.

With respect to fixed-priority assignments, dynamic scheduling algorithms are characterized by higher schedulability bounds, which allow the processor to be better utilized, increase the size of aperiodic servers, and enhance aperiodic responsiveness. Consider, for example, a set of two periodic tasks with the same utilization $U_1 = U_2 = 0.3$, so that $U_p = 0.6$. If priorities are assigned to periodic tasks based on RM and aperiodic requests are served by a Sporadic Server, the maximum server size that guarantees periodic schedulability is given by Equation (5.24) and is $U_{SS}^{max} = 2/P - 1$, where $P = (U_1 + 1)(U_2 + 1) = 1.69$. Hence, we have $U_{SS}^{max} \simeq 0.18$. On the other hand, if periodic tasks are scheduled by EDF, the processor utilization bound goes up to 1.0, so the maximum server size can be increased up to $U_s = 1 - U_p = 0.4$.

For the sake of clarity, all properties of the algorithms presented in this chapter are proven under the following assumptions:

---

[1] Part of this chapter is taken from the paper "Scheduling Aperiodic Tasks in Dynamic Priority Systems" by M. Spuri and G. Buttazzo, published in *Real-Time Systems*, 10(2), March 1996.

- All periodic tasks $\tau_i : i = 1, \ldots, n$ have hard deadlines and their schedulability must be guaranteed off line.

- All aperiodic tasks $J_i : i = 1, \ldots, m$ do not have deadlines and must be scheduled as soon as possible, but without jeopardizing the schedulability of the periodic tasks.

- Each periodic task τ_i has a period T_i, a computation time C_i, and a relative deadline D_i equal to its period.

- All periodic tasks are simultaneously activated at time $t = 0$.

- Each aperiodic task has a known computation time but an unknown arrival time.

Some of the assumptions listed above can easily be relaxed to handle periodic tasks with arbitrary phasing and relative deadlines different from their periods. Shared resources can also be included in the model assuming an access protocol like the Stack Resource Policy [Bak91]. In this case, the schedulability analysis has to be consequently modified to take into account the blocking factors due to the mutually exclusive access to resources. For some algorithms, the possibility of handling firm aperiodic tasks is also discussed.

The rest of the chapter is organized as follows. In the next two sections, we discuss how two fixed-priority service algorithms – namely, the Priority Exchange and the Sporadic Server algorithms – can be extended to work under the EDF priority assignment. Then, we introduce three new aperiodic service algorithms, based on dynamic deadline assignments, that greatly improve the performance of the previous fixed-priority extensions. One of these algorithms, the EDL server, is shown to be optimal, in the sense that it minimizes the average response time of aperiodic requests.

6.2   DYNAMIC PRIORITY EXCHANGE SERVER

The *Dynamic Priority Exchange* (DPE) server is an aperiodic service technique proposed by Spuri and Buttazzo [SB94, SB96] that can be viewed as an extension of the Priority Exchange server [LSS87], adapted to work with a deadline-based scheduling algorithm. The main idea of the algorithm is to let the server trade its runtime with the runtime of lower-priority periodic tasks (under EDF this means a longer deadline) in case there are no aperiodic requests pending. In this way, the server runtime is only exchanged with periodic tasks but never wasted (unless there are idle times). It is simply preserved, even if at a lower priority, and it can be later reclaimed when aperiodic requests enter the system.

The algorithm is defined as follows:

- The DPE server has a specified period T_s and a capacity C_s.

- At the beginning of each period, the server's *aperiodic* capacity is set to C_S^d, where d is the deadline of the current server period.

- Each deadline d associated to the instances (completed or not) of the ith periodic task has an aperiodic capacity, $C_{S_i}^d$, initially set to 0.

- Aperiodic capacities (those greater than 0) receive priorities according to their deadlines and the EDF algorithm, like all the periodic task instances (ties are broken in favor of capacities; that is, aperiodic requests).

- Whenever the highest-priority entity in the system is an aperiodic capacity of C units of time the following happens:

    - if there are aperiodic requests in the system, these are served until they complete or the capacity is exhausted (each request consumes a capacity equal to its execution time);

    - if there are no aperiodic requests pending, the periodic task having the shortest deadline is executed; a capacity equal to the length of the execution is added to the aperiodic capacity of the task deadline and is subtracted from C (that is, the deadlines of the highest-priority capacity and the periodic task are exchanged);

    - if neither aperiodic requests nor periodic task instances are pending, there is an idle time and the capacity C is consumed until, at most, it is exhausted.

An example of schedule produced by the DPE algorithm is illustrated in Figure 6.1. Two periodic tasks, τ_1 and τ_2, with periods $T_1 = 8$ and $T_2 = 12$ and worst-case execution times $C_1 = 2$ and $C_2 = 3$, and a DPE server with period $T_s = 6$ and capacity $C_s = 3$, are present in the system.

At time $t = 0$, the aperiodic capacities $C_{S_1}^8$ (with deadline 8) and $C_{S_2}^{12}$ (with deadline 12) are set to 0, while the server capacity (with deadline 6) is set to $C_s = C_S^6 = 3$. Since no aperiodic requests are pending, the two first periodic instances of τ_1 and τ_2 are executed and C_s is consumed in the first three units of time. In the same interval, two units of time are accumulated in $C_{S_1}^8$ and one unit in $C_{S_2}^{12}$.

At time $t = 3$, $C_{S_1}^8$ is the highest-priority entity in the system. Again, since no aperiodic requests are pending, τ_2 keeps executing and the two units of $C_{S_1}^8$ are consumed and accumulated in $C_{S_2}^{12}$. In the following three units of time the processor is idle and
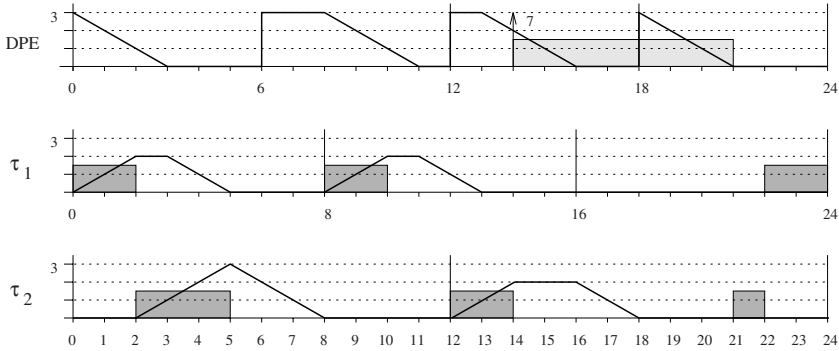
Figure 6.1 Dynamic Priority Exchange server example.

$C_{S_2}^{12}$ is completely consumed. Note that at time $t = 6$ the server capacity $C_s = C_S^{12}$ is set at value 3 and is preserved until time $t = 8$, when it becomes the highest-priority entity in the system (ties among aperiodic capacities are assumed to be broken in a FIFO order). At time $t = 8$, two units of C_S^{12} are exchanged with $C_{S_1}^{16}$, while the third unit of the server is consumed since the processor is idle.

At time $t = 14$, an aperiodic request, J_1, of seven units of time enters the system. Since $C_S^{18} = 2$, the first two units of J_1 are served with deadline 18, while the next two units are served with deadline 24, using the capacity $C_{S_2}^{24}$. Finally, the last three units are also served with deadline 24 because at time $t = 18$ the server capacity C_S^{24} is set to 3.

6.2.1   SCHEDULABILITY ANALYSIS

The schedulability condition for a set of periodic tasks scheduled together with a DPE server is now analyzed. Intuitively, the server behaves like any other periodic task. The difference is that it can trade its runtime with the runtime of lower-priority tasks. When a certain amount of time is traded, one or more lower-priority tasks are run at a higher-priority level, and their lower-priority time is preserved for possible aperiodic requests. This run-time exchange, however, does not affect schedulability; thus, the periodic task set can be guaranteed using the classical Liu and Layland condition:

$$U_p + U_s \leq 1,$$

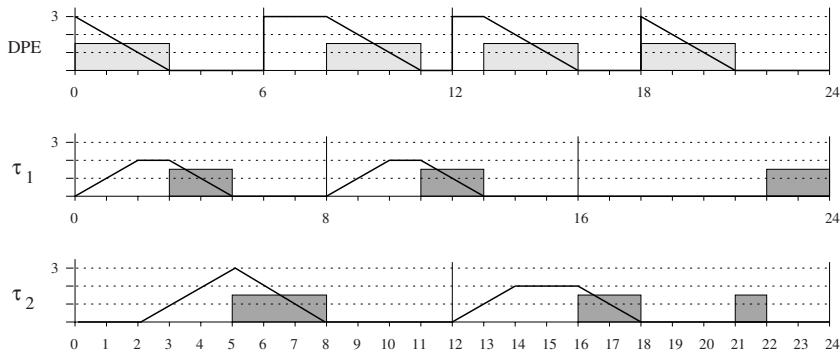where U_p is the utilization factor of the periodic tasks and U_s is the utilization factor of the DPE server.

Figure 6.2 DPE server schedulability.

In order to prove this result, given a schedule σ produced using the DPE algorithm, consider a schedule σ' built in the following way:

- Replace the DPE server with a periodic task τ_s with period T_s and worst-case execution time C_s, so that in σ' τ_s executes whenever the server capacity is consumed in σ.

- The execution of periodic instances during deadline exchanges is postponed until the capacity decreases.

- All other executions of periodic instances are left as in σ.

Note that from the definition of the DPE algorithm, at any time, at most one aperiodic capacity decreases in σ, so σ' is well defined. Also observe that, in each feasible schedule produced by the DPE algorithm, all the aperiodic capacities are exhausted before their respective deadlines.

Figure 6.2 shows the schedule σ' obtained from the schedule σ of Figure 6.1. Note that all the periodic executions corresponding to increasing aperiodic capacities have been moved to the corresponding intervals in which the same capacities decrease. Also note that the schedule σ' does not depend on the aperiodic requests but depends only on the characteristics of the server and on the periodic task set. Based on this observation, the following theorem can be proved:

Theorem 6.1 (Spuri, Buttazzo) *Given a set of periodic tasks with processor utilization U_p and a DPE server with processor utilization U_s, the whole set is schedulable by EDF* if *and* only if

$$U_p + U_s \leq 1.$$

Proof. For any aperiodic load, all the schedules produced by the DPE algorithm have a unique corresponding EDF schedule σ', built according to the definition given above. Moreover, the task set in σ' is periodic with a processor utilization $U = U_p + U_s$. Hence, σ' is feasible if and only if $U_p + U_s \leq 1$. Now we show that σ is feasible if and only if σ' is feasible.

Observe that in each schedule σ the completion time of a periodic instance is always less than or equal to the completion time of the corresponding instance in the schedule σ'. Hence, if σ' is feasible, then also σ is feasible; that is, the periodic task set is schedulable with the DPE algorithm. Vice versa, observing that σ' is a particular schedule produced by the DPE algorithm when there are enough aperiodic requests, if σ is feasible, then σ' will also be feasible; hence, the theorem holds. □

6.2.2   RECLAIMING SPARE TIME

In hard real-time systems, the guarantee test of critical tasks is done by performing a worst-case schedulability analysis; that is, assuming the maximum execution time for all task instances. However, when such a peak load is not reached because the actual execution times are less than the worst-case values, it is not always obvious how to reclaim the spare time efficiently.

Using a DPE server, the spare time unused by periodic tasks can be easily reclaimed for servicing aperiodic requests. Whenever a periodic task completes, it is sufficient to add its spare time to the corresponding aperiodic capacity. An example of reclaiming mechanism is shown in Figure 6.3.

As it can be seen from the capacity plot, at the completion time of the first two periodic instances, the corresponding aperiodic capacities ($C_{S_1}^8$ and $C_{S_2}^{12}$) are incremented by an amount equal to the spare time saved. Thanks to this reclaiming mechanism, the first aperiodic request can receive immediate service for all the seven units of time required, completing at time $t = 11$. Without reclaiming, the request would complete at time $t = 12$.

Note that reclaiming the spare time of periodic tasks as aperiodic capacities does not affect the schedulability of the system. In fact, any spare time is already "allocated" to a priority level corresponding to its deadline when the task set has been guaranteed. Hence, the spare time can be used safely if requested with the same deadline.
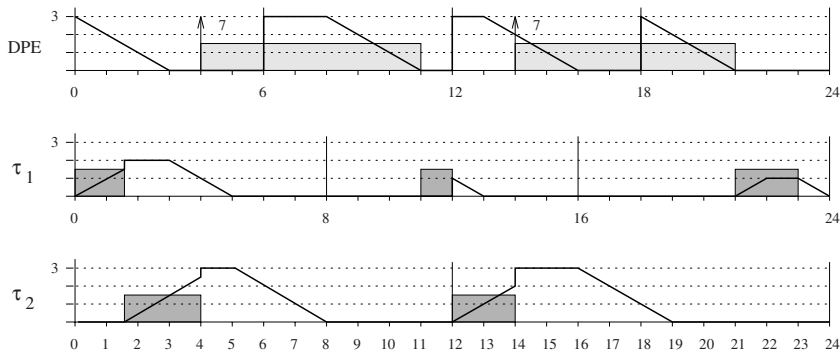
Figure 6.3 DPE server resource reclaiming.

6.3 DYNAMIC SPORADIC SERVER

The *Dynamic Sporadic Server*[2] (DSS) is an aperiodic service strategy proposed by Spuri and Buttazzo [SB94, SB96] that extends the Sporadic Server [SSL89] to work under a dynamic EDF scheduler. Similarly to other servers, DSS is characterized by a period T_s and a capacity C_s, which is preserved for possible aperiodic requests. Unlike other server algorithms, however, the capacity is not replenished at its full value at the beginning of each server period but only when it has been consumed. The times at which the replenishments occur are chosen according to a replenishment rule, which allows the system to achieve full processor utilization.

The main difference between the classical SS and its dynamic version consists in the way the priority is assigned to the server. Whereas SS has a fixed priority chosen according to the RM algorithm (that is, according to its period T_s), DSS has a dynamic priority assigned through a suitable deadline. The deadline assignment and the capacity replenishment are defined by the following rules:

- When the server is created, its capacity C_s is initialized at its maximum value.

- The next replenishment time RT and the current server deadline d_s are set as soon as $C_s > 0$ and there is an aperiodic request pending. If t_A is such a time, then $RT = d_s = t_A + T_s$.

- The replenishment amount RA to be done at time RT is computed when the last aperiodic request is completed or C_s has been exhausted. If t_I is such a time, then RA is set equal to the capacity consumed within the interval $[t_A, t_I]$.

---

[2]A similar algorithm called Deadline Sporadic Server has been independently developed by Ghazalie and Baker in [GB95].
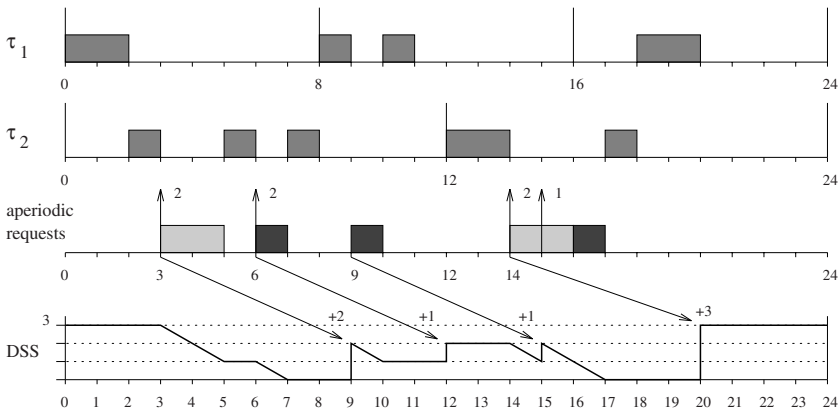
Figure 6.4   Dynamic Sporadic Server example.

Figure 6.4 illustrates an EDF schedule obtained on a task set consisting of two periodic tasks with periods $T_1 = 8$, $T_2 = 12$ and execution times $C_1 = 2$, $C_2 = 3$, and a DSS with period $T_s = 6$ and capacity $C_s = 3$.

At time $t = 0$, the server capacity is initialized at its full value $C_s = 3$. Since there are no aperiodic requests pending, the processor is assigned to task τ_1, which has the earliest deadline. At time $t = 3$, an aperiodic request with execution time 2 arrives and, since $C_s > 0$, the first replenishment time and the server deadline are set to $RT_1 = d_s = 3 + T_s = 9$. Being d_s the earliest deadline, DSS becomes the highest-priority task in the system and the request is serviced until completion. At time $t = 5$, the request is completed and no other aperiodic requests are pending; hence, a replenishment of two units of time is scheduled to occur at time $RT_1 = 9$.

At time $t = 6$, a second aperiodic requests arrives. Being $C_s > 0$, the next replenishment time and the new server deadline are set to $RT_2 = d_s = 6 + T_s = 12$. Again, the server becomes the highest-priority task in the system (we assume that ties among tasks are always resolved in favor of the server) and the request receives immediate service. This time, however, the capacity has only one unit of time available, and it gets exhausted at time $t = 7$. Consequently, a replenishment of one unit of time is scheduled for $RT_2 = 12$, and the aperiodic request is delayed until $t = 9$, when C_s becomes again greater than zero. At time $t = 9$, the next replenishment time and the new deadline of the server are set to $RT_3 = d_s = 9 + T_s = 15$. As before, DSS becomes the highest-priority task; thus, the aperiodic request receives immediate service and finishes at time $t = 10$. A replenishment of one unit is then scheduled to occur at time $RT_3 = 15$.
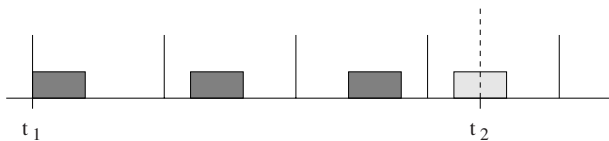
Figure 6.5 Computational demand of a periodic task in $[t_1, t_2]$.

Note that as long as the server capacity is greater than zero, all pending aperiodic requests are executed with the same deadline. In Figure 6.4 this happens at time $t = 14$, when the last two aperiodic requests are serviced with the same deadline $d_s = 20$.

6.3.1 SCHEDULABILITY ANALYSIS

To prove the schedulability bound for the Dynamic Sporadic Server, we first show that the server behaves like a periodic task with period T_s and execution time C_s.

Given a periodic task τ_i, we first note that in any generic interval $[t_1, t_2]$ such that τ_i is released at t_1, the computation time scheduled by EDF with deadline less than or equal to t_2 is such that (see Figure 6.5)

$$C_i(t_1, t_2) \leq \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i.$$

The following Lemma shows that the same property is true for DSS.

Lemma 6.1 *In each interval of time $[t_1, t_2]$, such that t_1 is the time at which DSS becomes ready (that is, an aperiodic request arrives and no other aperiodic requests are being served), the maximum aperiodic time executed by DSS in $[t_1, t_2]$ satisfies the following relation:*

$$C_{ape} \leq \left\lfloor \frac{t_2 - t_1}{T_s} \right\rfloor C_s.$$

Proof. Since replenishments are always equal to the time consumed, the server capacity is at any time less than or equal to its initial value. Also, the replenishment policy establishes that the consumed capacity cannot be reclaimed before T_s units of time after the instant at which the server has become ready. This means that, from the time t_1 at which the server becomes ready, at most C_s time can be consumed in each subsequent interval of time of length T_s; hence, the thesis follows. □

Given that DSS behaves like a periodic task, the following theorem states that a full processor utilization is still achieved.

Theorem 6.2 (Spuri, Buttazzo) *Given a set of n periodic tasks with processor utilization U_p and a Dynamic Sporadic Server with processor utilization U_s, the whole set is schedulable* if *and* only if

$$U_p + U_s \leq 1.$$

Proof. *If.* Assume $U_p + U_s \leq 1$ and suppose there is an overflow at time t. The overflow is preceded by a period of continuous utilization of the processor. Furthermore, from a certain point t' on ($t' < t$), only instances of tasks ready at t' or later and having deadlines less than or equal to t are run (the server may be one of these tasks). Let C be the total execution time demanded by these instances. Since there is an overflow at time t, we must have $t - t' < C$. We also know that

$$
\begin{aligned}
C &\leq \sum_{i=1}^{n} \left\lfloor \frac{t-t'}{T_i} \right\rfloor C_i + C_{ape} \\
&\leq \sum_{i=1}^{n} \left\lfloor \frac{t-t'}{T_i} \right\rfloor C_i + \left\lfloor \frac{t-t'}{T_s} \right\rfloor C_s \\
&\leq \sum_{i=1}^{n} \frac{t-t'}{T_i} C_i + \frac{t-t'}{T_s} C_s \\
&\leq (t-t')(U_p + U_s).
\end{aligned}
$$

Thus, it follows that

$$U_p + U_s > 1,$$

a contradiction.

Only If. Since DSS behaves as a periodic task with period T_s and execution time C_s, the server utilization factor is $U_s = C_s/T_s$ and the total utilization factor of the processor is $U_p + U_s$. Hence, if the whole task set is schedulable, from the EDF schedulability bound [LL73] we can conclude that $U_p + U_s \leq 1$. $\square$

6.4 TOTAL BANDWIDTH SERVER

Looking at the characteristics of the Sporadic Server algorithm, it can be easily seen that, when the server has a long period, the execution of the aperiodic requests can be delayed significantly. This is due to the fact that when the period is long, the server is always scheduled with a far deadline. And this is regardless of the aperiodic execution times.

There are two possible approaches to reduce the aperiodic response times. The first is, of course, to use a Sporadic Server with a shorter period. This solution, however, increases the run-time overhead of the algorithm because, to keep the server utilization constant, the capacity has to be reduced proportionally, but this causes more frequent replenishments and increases the number of context switches with the periodic tasks.

A second approach, less obvious, is to assign a possible earlier deadline to each aperiodic request. The assignment must be done in such a way that the overall processor utilization of the aperiodic load never exceeds a specified maximum value U_s. This is the main idea behind the *Total Bandwidth Server* (TBS), a simple and efficient aperiodic service mechanism proposed by Spuri and Buttazzo [SB94, SB96]. The name of the server comes from the fact that, each time an aperiodic request enters the system, the total bandwidth of the server is immediately assigned to it, whenever possible.

In particular, when the kth aperiodic request arrives at time $t = r_k$, it receives a deadline
$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s},$$
where C_k is the execution time of the request and U_s is the server utilization factor (that is, its bandwidth). By definition $d_0 = 0$. Note that in the deadline assignment rule the bandwidth allocated to previous aperiodic requests is considered through the deadline d_{k-1}.

Once the deadline is assigned, the request is inserted into the ready queue of the system and scheduled by EDF as any other periodic instance. As a consequence, the implementation overhead of this algorithm is practically negligible.

Figure 6.6 shows an example of an EDF schedule produced by two periodic tasks with periods $T_1 = 6$, $T_2 = 8$ and execution times $C_1 = 3$, $C_2 = 2$, and a TBS with utilization $U_s = 1 - U_p = 0.25$. The first aperiodic request arrives at time $t = 3$ and is serviced with deadline $d_1 = r_1 + C_1/U_s = 3 + 1/0.25 = 7$. Being d_1 the earliest deadline in the system, the aperiodic request is executed immediately. Similarly, the second request, which arrives at time $t = 9$, receives a deadline $d_2 = r_2 + C_2/U_s = 17$, but it is not serviced immediately, since at time $t = 9$ there is an active periodic
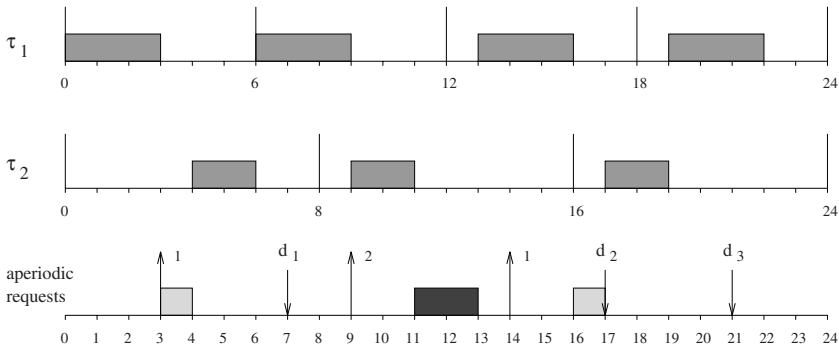
Figure 6.6 Total Bandwidth Server example.

task, τ_2, with a shorter deadline, equal to 16. Finally, the third aperiodic request arrives at time $t = 14$ and gets a deadline $d_3 = \max(r_3, d_2) + C_3/U_s = 21$. It does not receive immediate service, since at time $t = 14$ task τ_1 is active and has an earlier deadline (18).

6.4.1 SCHEDULABILITY ANALYSIS

In order to derive a schedulability test for a set of periodic tasks scheduled by EDF in the presence of a TBS, we first show that the aperiodic load executed by TBS cannot exceed the utilization factor U_s defined for the server.

Lemma 6.2 *In each interval of time $[t_1, t_2]$, if C_{ape} is the total execution time demanded by aperiodic requests arrived at t_1 or later and served with deadlines less than or equal to t_2, then*

$$C_{ape} \leq (t_2 - t_1)U_s.$$

Proof. By definition

$$C_{ape} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k.$$

Given the deadline assignment rule of the TBS, there must exist two aperiodic requests with indexes k_1 and k_2 such that

$$\sum_{t_1 \leq r_k, d_k \leq t_2} C_k = \sum_{k=k_1}^{k_2} C_k.$$

It follows that

$$
\begin{aligned}
C_{ape} &= \sum_{k=k_1}^{k_2} C_k \\
&= \sum_{k=k_1}^{k_2} [d_k - \max(r_k, d_{k-1})]U_s \\
&\leq [d_{k_2} - \max(r_{k_1}, d_{k_1-1})]U_s \\
&\leq (t_2 - t_1)U_s.
\end{aligned}
$$

□

The main result on TBS schedulability can now be proved.

Theorem 6.3 (Spuri, Buttazzo) *Given a set of n periodic tasks with processor utilization U_p and a TBS with processor utilization U_s, the whole set is schedulable by EDF if* and *only if*

$$
U_p + U_s \leq 1.
$$

Proof. *If.* Assume $U_p + U_s \leq 1$ and suppose there is an overflow at time t. The overflow is preceded by a period of continuous utilization of the processor. Furthermore, from a certain point t' on ($t' < t$), only instances of tasks ready at t' or later and having deadlines less than or equal to t are run. Let C be the total execution time demanded by these instances. Since there is an overflow at time t, we must have

$$
t - t' < C.
$$

We also know that

$$
\begin{aligned}
C &\leq \sum_{i=1}^{n} \left\lfloor \frac{t - t'}{T_i} \right\rfloor C_i + C_{ape} \\
&\leq \sum_{i=1}^{n} \frac{t - t'}{T_i} C_i + (t - t')U_s \\
&\leq (t - t')(U_p + U_s).
\end{aligned}
$$

Thus, it follows that

$$
U_p + U_s > 1,
$$

a contradiction.

Only If. If an aperiodic request enters the system periodically, with period T_s and execution time $C_s = T_s U_s$, the server behaves exactly as a periodic task with period T_s and execution time C_s, and the total utilization factor of the processor is $U_p + U_s$. Hence, if the whole task set is schedulable, from the EDF schedulability bound [LL73] we can conclude that $U_p + U_s \leq 1$. □

6.5   EARLIEST DEADLINE LATE SERVER

The Total Bandwidth Server is able to provide good aperiodic responsiveness with extreme simplicity. However, a better performance can still be achieved through more complex algorithms. For example, looking at the schedule in Figure 6.6, we could argue that the second and the third aperiodic requests may be served as soon as they arrive, without compromising the schedulability of the system. This is possible because, when the requests arrive, the active periodic instances have enough slack time (laxity) to be safely preempted.

Using the available slack of periodic tasks for advancing the execution of aperiodic requests is the basic principle adopted by the EDL server [SB94, SB96]. This aperiodic service algorithm can be viewed as a dynamic version of the Slack Stealing algorithm [LRT92].

The definition of the EDL server makes use of some results presented by Chetto and Chetto [CC89]. In this paper, two complementary versions of EDF – namely, EDS and EDL – are proposed. Under EDS the active tasks are processed as soon as possible, whereas under EDL they are processed as late as possible. An important property of EDL is that in any interval $[0, t]$ it guarantees the maximum available idle time. In the original paper, this result is used to build an acceptance test for aperiodic tasks with hard deadlines, while here it is used to build an optimal server mechanism for soft aperiodic activities.

To simplify the description of the EDL server, $\omega_J^A(t)$ denotes the following *availability function*, defined for a scheduling algorithm A and a task set J:

$$\omega_J^A(t) = \begin{cases} 1 & \text{if the processor is idle at } t \\ 0 & \text{otherwise.} \end{cases}$$
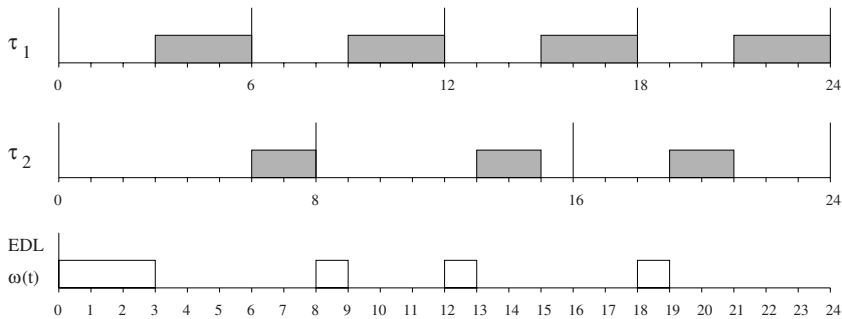
Figure 6.7 Availability function under EDL.

The integral of $\omega_J^A(t)$ on an interval of time $[t_1, t_2]$ is denoted by $\Omega_J^A(t_1, t_2)$ and gives the total idle time in the specified interval. The function ω_J^{EDL} for the task set of Figure 6.6 is depicted in Figure 6.7.

The result of optimality addressed above is stated by Chetto and Chetto [CC89] in Theorem 2, which we recall here.

Theorem 6.4 (Chetto and Chetto) *Let J be any aperiodic task set and A any pre-emptive scheduling algorithm. For any instant t,*

$$\Omega_J^{\text{EDL}}(0, t) \geq \Omega_J^A(0, t).$$

This result allows to develop an optimal server using the idle times of an EDL scheduler. In particular, given a periodic task set J, the function ω_J^A, which is periodic with *hyperperiod* $H = \text{lcm}(T_1, \ldots, T_n)$, can be represented by means of two arrays. The first, $\mathcal{E} = (e_0, e_1, \ldots, e_p)$, represents the times at which idle times occur, while the second, $\mathcal{D} = (\Delta_0, \Delta_1, \ldots, \Delta_p)$, represents the lengths of these idle times. The two arrays for the example in Figure 6.7 are shown in Table 6.1. Note that idle times occur only after the release of a periodic task instance.

i	0	1	2	3
e_i	0	8	12	18
Δ_i	3	1	1	1

Table 6.1 Idle times under EDL.
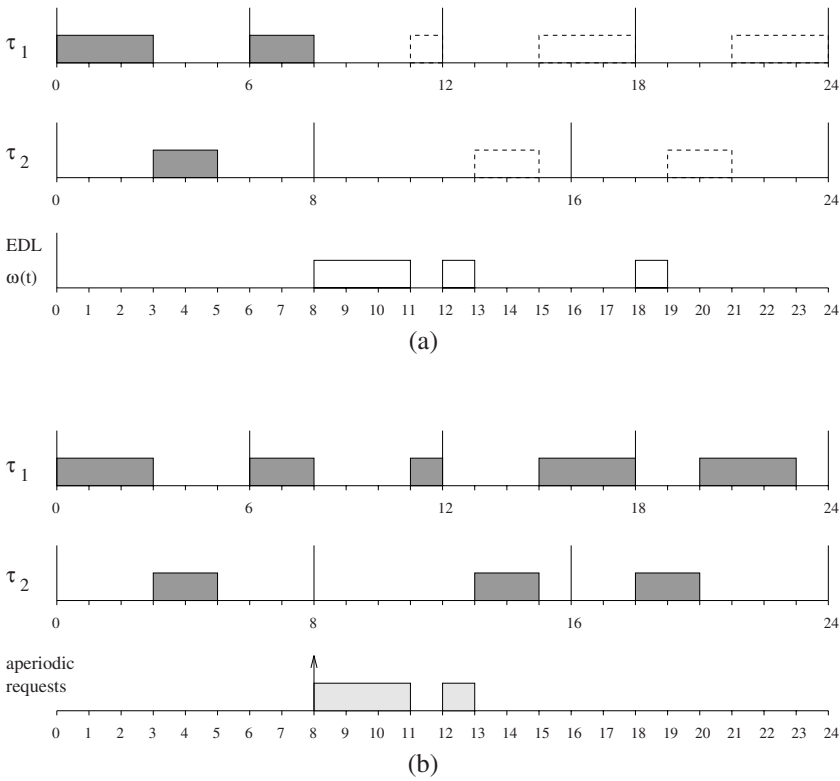
Figure 6.8   a. Idle times available at time $t = 8$ under EDL. **b.** Schedule of the aperiodic request with the EDL server.

The basic idea behind the EDL server is to use the idle times of an EDL schedule to execute aperiodic requests as soon as possible. When there are no aperiodic activities in the system, periodic tasks are scheduled according to the EDF algorithm. Whenever a new aperiodic request enters the system (and no previous aperiodic is still active) the idle times of an EDL scheduler applied to the current periodic task set are computed and then used to schedule the aperiodic requests pending. Figure 6.8 shows an example of the EDL service mechanism.

Here, an aperiodic request with an execution time of 4 units arrives at time $t = 8$. The idle times of an EDL schedule are recomputed using the current periodic tasks, as shown in Figure 6.8a. The request is then scheduled according to the computed idle times (Figure 6.8b). Notice that the server automatically allocates a bandwidth $1 - U_p$

to aperiodic requests. The response times achieved by this method are optimal, so they cannot be reduced further.

The procedure to compute the idle times of the EDL schedule is described in Chetto and Chetto [CC89] and is not reported here. However, it is interesting to observe that not all the idle times have to be recomputed, but only those preceding the deadline of the current active periodic task with the longest deadline.

The worst-case complexity of the algorithm is $O(Nn)$, where n is the number of periodic tasks and N is the number of distinct periodic requests that occur in the hyperperiod. In the worst case, N can be very large and, hence, the algorithm may be of little practical interest. As for the Slack Stealer, the EDL server will be used to provide a lower bound to the aperiodic response times and to build a nearly optimal implementable algorithm, as described in the next section.

6.5.1   EDL SERVER PROPERTIES

The schedulability analysis of the EDL server is quite straightforward. In fact, all aperiodic activities are executed using the idle times of a particular EDF schedule; thus, the feasibility of the periodic task set cannot be compromised. This is stated in the following theorem:

Theorem 6.5 (Spuri, Buttazzo) *Given a set of n periodic tasks with processor utilization U_p and the corresponding EDL server (whose behavior strictly depends on the characteristics of the periodic task set), the whole set is schedulable* if *and* only if

$$U_p \leq 1.$$

Proof. *If.* Since the condition ($U_p \leq 1$) is sufficient for guaranteeing the schedulability of a periodic task set under EDF, it is also sufficient under EDL, which is a particular implementation of EDF. The algorithm schedules the periodic tasks according to one or the other implementation, depending on the particular sequence of aperiodic requests. When aperiodic requests are pending, they are scheduled during precomputed idle times of the periodic tasks. In both cases the timeliness of the periodic task set is unaffected and no deadline is missed.

Only If. If a periodic task set is schedulable with an EDL server, it will be also schedulable without the EDL server, and hence ($U_p \leq 1$). □

We finally show that the EDL server is optimal; that is, the response times of the aperiodic requests under the EDL algorithm are the best achievable.

Lemma 6.3 *Let A be any online preemptive algorithm, τ a periodic task set, and J_i an aperiodic request. If $f^A_{\tau \cup \{J_i\}}(J_i)$ is the finishing time of J_i when $\tau \cup \{J_i\}$ is scheduled by A, then*

$$f^{\text{EDL server}}_{\tau \cup \{J_i\}}(J_i) \ \leq \ f^A_{\tau \cup \{J_i\}}(J_i).$$

Proof.   Suppose J_i arrives at time t, and let $\tau(t)$ be the set of the current active periodic instances (ready but not yet completed) and the future periodic instances. The new task J_i is scheduled together with the tasks in $\tau(t)$. In particular, consider the schedule σ of $\tau \cup \{J_i\}$ under A. Let A' be another algorithm that schedules the tasks in $\tau(t)$ at the same time as in σ, and σ' be the corresponding schedule. J_i is executed during some idle periods of σ'. Applying Theorem 6.4 with the origin of the time axis translated to t (this can be done since A is online), we know that for each $t' \geq t$

$$\Omega^{\text{EDL}}_{\tau(t)}(t, t') \ \geq \ \Omega^{A'}_{\tau(t)}(t, t').$$

Recall that when there are aperiodic requests, the EDL server allocates the executions exactly during the idle times of EDL. Being

$$\Omega^{\text{EDL}}_{\tau(t)}(t, f^{\text{EDL server}}_{\tau \cup \{J_i\}}(J_i)) \ \geq \ \Omega^{A'}_{\tau(t)}(t, f^{\text{EDL server}}_{\tau \cup \{J_i\}}(J_i))$$

it follows that

$$f^{\text{EDL}}_{\tau \cup \{J_i\}}(J_i) \ \leq \ f^A_{\tau \cup \{J_i\}}(J_i).$$

That is, under the EDL server, J_i is never completed later than under the A algorithm.
□

6.6   IMPROVED PRIORITY EXCHANGE SERVER

Although optimal, the EDL server has too much overhead to be considered practical. However, its main idea can be usefully adopted to develop a less complex algorithm that still maintains a nearly optimal behavior.

The heavy computation of the idle times can be avoided by using the mechanism of priority exchanges. With this mechanism, in fact, the system can easily keep track of the time advanced to periodic tasks and possibly reclaim it at the right priority level.

The idle times of the EDL algorithm can be precomputed off-line and the server can use them to schedule aperiodic requests, when there are any, or to advance the execution of periodic tasks. In the latter case, the idle time advanced can be saved as aperiodic capacity at the priority levels of the periodic tasks executed.

The idea described above is used by the algorithm called *Improved Priority Exchange* (IPE) [SB94, SB96]. In particular, the DPE server is modified using the idle times of an EDL scheduler. There are two main advantages in this approach. First, a far more efficient replenishment policy is achieved for the server. Second, the resulting server is no longer periodic, and it can always run at the highest priority in the system. The IPE server is thus defined in the following way:

- The IPE server has an aperiodic capacity, initially set to 0.

- At each instant $t = e_i + kH$, with $0 \leq i \leq p$ and $k \geq 0$, a replenishment of Δ_i units of time is scheduled for the server capacity; that is, at time $t = e_0$ the server will receive Δ_0 units of time (the two arrays $\mathcal{E}$ and $\mathcal{D}$ have been defined in the previous section).

- The server priority is always the highest in the system, regardless of any other deadline.

- All other rules of IPE (aperiodic requests and periodic instances executions, exchange and consumption of capacities) are the same as for a DPE server.

The same task set of Figure 6.8 is scheduled with an IPE server in Figure 6.9.

Note that the server replenishments are set according to the function ω_τ^{EDL}, illustrated in Figure 6.7.

When the aperiodic request arrives at time $t = 8$, one unit of time is immediately allocated to it by the server. However, other two units are available at the priority level corresponding to the deadline 12, due to previous deadline exchanges, and are allocated right after the first one. The last one is allocated later, at time $t = 12$, when the server receives a further unit of time. In this situation, the optimality of the response time is kept. As we will show later, there are only rare situations in which the optimal EDL server performs slightly better than IPE. That is, IPE almost always exhibits a nearly optimal behavior.
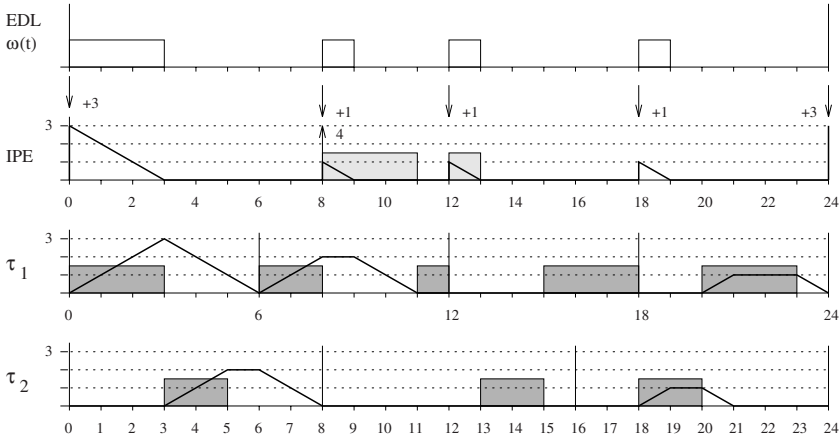
Figure 6.9   Improved Priority Exchange server example.

6.6.1   SCHEDULABILITY ANALYSIS

In order to analyze the schedulability of an IPE server, it is useful to define a transformation among schedules similar to that defined for the DPE server. In particular, given a schedule σ produced by the IPE algorithm, we build the schedule σ' in the following way:

■   Each execution of periodic instances during deadline exchanges (that is, increase in the corresponding aperiodic capacity) is postponed until the capacity decreases.

■   All other executions of periodic instances are left as in σ.

In this case, the server is not substituted with another task. Again σ' is well defined and is invariant; that is, it does not depend on σ but only on the periodic task set. Moreover, σ' is the schedule produced by EDL applied to the periodic task set (compare Figure 6.7 with Figure 6.9). The optimal schedulability is stated by the following theorem:

Theorem 6.6 (Spuri, Buttazzo) *Given a set of n periodic tasks with processor utilization U_p and the corresponding IPE server (the parameters of the server depend on the periodic task set), the whole set is schedulable* if *and* only if

$$U_p \leq 1$$

(the server automatically allocates the bandwidth $1 - U_p$ to aperiodic requests).

Proof. *If.* The condition is sufficient for the schedulability of the periodic task set under EDF, thus even under EDL, which is a particular implementation of EDF. Now, observe that in each schedule produced by the IPE algorithm the completion times of the periodic instances are never greater than the completion times of the corresponding instances in σ', which is the schedule of the periodic task set under EDL. That is, no periodic instance can miss its deadline. The thesis follows.

Only If. Trivial, since the condition is necessary even for the periodic task set only. $\square$

6.6.2   REMARKS

The reclaiming of unused periodic execution time can be done in the same way as for the DPE server. When a periodic task completes, its spare time is added to the corresponding aperiodic capacity. Again, this behavior does not affect the schedulability of the system. The reason is of course the same as for the DPE server.

To implement the IPE server, the two arrays $\mathcal{E}$ and $\mathcal{D}$ must be precomputed before the system is run. The replenishments of the server capacity are no longer periodic, but this does not change the complexity, which is comparable with that of DPE. What can change dramatically is the memory requirement. In fact, if the periods of periodic tasks are not harmonically related, we could have a huge *hyperperiod* $H = \text{lcm}(T_1, \ldots, T_n)$, which would require a great memory space to store the two arrays $\mathcal{E}$ and $\mathcal{D}$.

6.7   IMPROVING TBS

The deadline assignment rule used by the TBS algorithm is a simple and efficient technique for servicing aperiodic requests in a hard real-time periodic environment. At the cost of a slightly higher complexity, such a rule can be modified to enhance aperiodic responsiveness. The key idea is to shorten the deadline assigned by the TBS as much as possible, still maintaining the periodic tasks schedulable [BS99].

If d_k is the deadline assigned to an aperiodic request by the TBS, a new deadline d'_k can be set at the estimated worst-case finishing time f_k of that request, scheduled by EDF with deadline d_k. The following lemma shows that setting the new deadline d'_k at the current estimated worst-case finishing time does not jeopardize schedulability:

Lemma 6.4 *Let σ be a feasible schedule of task set $\mathcal{T}$, in which an aperiodic job J_k is assigned a deadline d_k, and let f_k be the finishing time of J_k in σ. If d_k is substituted with $d'_k = f_k$, then the new schedule σ' produced by EDF is still feasible.*

Proof. Since σ remains feasible after d_k is substituted with $d'_k = f_k$ and all other deadlines are unchanged, the optimality of EDF [Der74] guarantees that σ' is also feasible. $\square$

The process of shortening the deadline can be applied recursively to each new deadline, until no further improvement is possible, given that the schedulability of the periodic task set must be preserved. If d^s_k is the deadline assigned to the aperiodic request J_k at step s and f^s_k is the corresponding finishing time in the current EDF schedule (achieved with d^s_k), the new deadline d^{s+1}_k is set at time f^s_k. At each step, the schedulability of the task set is guaranteed by Lemma 6.4.

The algorithm stops either when $d^s_k = d^{s-1}_k$ or after a maximum number of steps defined by the system designer for bounding the complexity. Note that the exact evaluation of f^s_k would require the development of the entire schedule up to the finishing time of request J_k, scheduled with d^s_k. However, there is no need to evaluate the exact value of f^s_k to shorten the deadline. Rather, the following upper bound can be used:

$$\tilde{f}^s_k = t + C^a_k + I_p(t, d^s_k), \tag{6.1}$$

where t is the current time (corresponding to the release time r_k of request J_k or to the completion time of the previous request), C^a_k is the worst-case computation time required by J_k, and $I_p(t, d^s_k)$ is the interference on J_k due to the periodic instances in the interval $[t, d^s_k)$. $\tilde{f}^s_k$ is an upper bound for f^s_k because it identifies the time at which J_k and all the periodic instances with deadline less than d^s_k end to execute. Hence, $f^s_k \leq \tilde{f}^s_k$.

The periodic interference $I_p(t, d^s_k)$ in Equation (6.1) can be expressed as the sum of two terms, $I_a(t, d^s_k)$ and $I_f(t, d^s_k)$, where $I_a(t, d^s_k)$ is the interference due to the currently active periodic instances with deadlines less than d^s_k, and $I_f(t, d^s_k)$ is the future interference due to the periodic instances activated after time t with deadline before d^s_k. Hence,

$$I_a(t, d^s_k) = \sum_{\tau_i \ active, \ d_i < d^s_k} c_i(t) \tag{6.2}$$

and

$$I_f(t, d^s_k) = \sum_{i=1}^{n} \max\left(0, \left\lceil \frac{d^s_k - next\_r_i(t)}{T_i} \right\rceil - 1\right) C_i, \tag{6.3}$$

where $next\_r_i(t)$ identifies the time greater than t at which the next periodic instance of task τ_i will be activated. If periodic tasks are synchronously activated at time zero, then

$$next\_r_i(t) = \left\lceil \frac{t}{T_i} \right\rceil T_i. \tag{6.4}$$

Since I_a and I_f can be computed in $O(n)$, the overall complexity of the deadline assignment algorithm is $O(Nn)$, where N is the maximum number of steps performed by the algorithm to shorten the initial deadline assigned by the TB server. We now show that $\tilde{f}_k^s$ is the real worst-case finishing time if it coincides with the deadline d_k^s.

Lemma 6.5 *In any feasible schedule, $\tilde{f}_k^s = f_k^s$ only if $\tilde{f}_k^s = d_k^s$.*

Proof. Assume that there exists a feasible schedule σ where $\tilde{f}_k^s = d_k^s$, but $\tilde{f}_k^s > f_k^s$. Since $\tilde{f}_k^s$ is the time at which J_k and all the periodic instances with deadline less than d_k^s end to execute, $\tilde{f}_k^s > f_k^s$ would imply that $\tilde{f}_k^s$ coincides with the end of a periodic instance having deadline less than $\tilde{f}_k^s = d_k^s$, meaning that this instance would miss its deadline. This is a contradiction; hence, the lemma follows. $\square$

6.7.1   AN EXAMPLE

The following example illustrates the deadline approximation algorithm. The task set consists of two periodic tasks, τ_1 and τ_2, with periods 3 and 4, and computation times 1 and 2, respectively. A single aperiodic job J_k arrives at time $t = 2$, requiring 2 units of computation time. The periodic utilization factor is $U_p = 5/6$, leaving a bandwidth of $U_s = 1/6$ for the aperiodic tasks.

When the aperiodic request arrives at time $t = 2$, it receives a deadline $d_k^0 = r_k + C_k^a/U_s = 14$, according to the TBS algorithm. The schedule produced by EDF using this deadline assignment is shown in Figure 6.10.

By applying Equations (6.2) and (6.3) we have

$$
\begin{aligned}
I_a(2, 14) &= c_2(2) &= 1 \\
I_f(2, 14) &= 3C_1 + 2C_2 &= 7,
\end{aligned}
$$

and, by Equation (6.1), we obtain

$$d_k^1 = \tilde{f}_k^0 = t + C_k^a + I_a + I_f = 12.$$
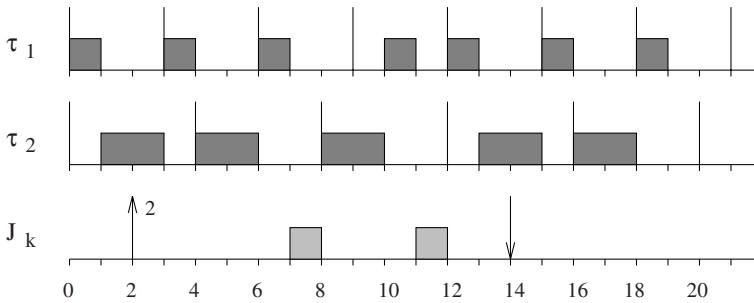
Figure 6.10   Schedule produced by EDF with $d_k^0 = 14$.

step	d_k^s	f_k^s
0	14	12
1	12	9
2	9	8
3	8	6
4	6	5
5	5	5

Table 6.2   Deadlines and finishing times computed by the algorithm.

In this case, it can easily be verified that the aperiodic task actually terminates at $t = 12$. This happens because the periodic tasks do not leave any idle time to the aperiodic task, which is thus compelled to execute at the end. Table 6.2 shows the subsequent deadlines evaluated at each step of the algorithm. In this example, six steps are necessary to find the shortest deadline for the aperiodic request.

The schedule produced by EDF using the shortest deadline $d_k^* = d_k^5 = 5$ is shown in Figure 6.11. Notice that at $t = 19$ the first idle time is reached, showing that the whole task set is schedulable.

6.7.2   OPTIMALITY

As far as the average case execution time of tasks is equal to the worst-case one, our deadline assignment method achieves optimality, yielding the minimum response time for each aperiodic task. Under this assumption, the following theorem holds.
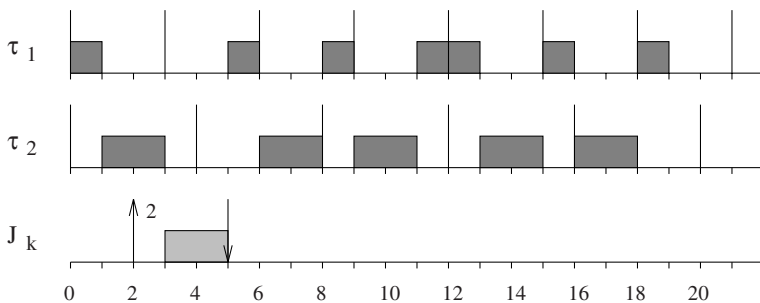
Figure 6.11 Schedule produced by EDF with $d_k^* = 5$.

Theorem 6.7 (Buttazzo, Sensini) *Let σ be a feasible schedule produced by EDF for a task set $\mathcal{T}$ and let f_k be the finishing time of an aperiodic request J_k, scheduled in σ with deadline d_k. If $f_k = d_k$, then $f_k = f_k^*$, where f_k^* is the minimum finishing time achievable by any other feasible schedule.*

Proof. Assume $f_k = d_k$, and let r_0 be the earliest request such that interval $[r_0, d_k]$ is fully utilized by J_k and by tasks with deadline less than d_k. Hence, in σ, d_k represents the time at which J_k and all instances with deadline less than d_k end to execute.

We show that any schedule σ' in which J_k finishes at $f_k' < d_k$ is not feasible. In fact, since $[r_0, d_k]$ is fully utilized and $f_k' < d_k$, in σ' d_k must be the finishing time of some periodic instance[3] with deadline less than d_k. As a consequence, σ' is not feasible. Thus, the theorem follows. □

6.8 PERFORMANCE EVALUATION

The algorithms described in this chapter have been simulated on a synthetic workload in order to compare the average response times achieved on soft aperiodic activities. For completeness, a dynamic version of the Polling Server has also been compared with the other algorithms.

The plots shown in Figure 6.12 have been obtained with a set of ten periodic tasks with periods ranging from 100 and 1000 units of time and utilization factor $U_p = 0.65$.

---

[3]Time d_k cannot be the finishing time of an aperiodic task, since we assume that aperiodic requests are served on a FCFS basis.

The aperiodic load was varied across the range of processor utilization unused by the periodic tasks, and in particular from 3% to 33%. The interarrival times for the aperiodic tasks were modeled using a Poisson arrival pattern, with average T_a, whereas the aperiodic computation times were modeled using an exponential distribution.

The processor utilization of the servers was set to all the utilization left by the periodic tasks; that is, $U_s = 1 - U_p$. The period of the periodic servers – namely Polling, DPE, and DSS – was set equal to the average aperiodic interarrival time (T_a) and, consequently, the capacity was set to $C_s = T_a U_s$.

In Figure 6.12, the performance of the algorithms is shown as a function of the aperiodic load. The load was varied by changing the average aperiodic service time, while the average interarrival time was set at the value of $T_a = 100$. Note that the data plotted for each algorithm represent the ratio of the average aperiodic response time relative to the response time of background service. In this way, an average response time equivalent to background service has a value of 1.0 on the graph. A value less than 1.0 corresponds to an improvement in the average aperiodic response time over background service. The lower the response time curve lies on these graphs, the better the algorithm is for improving aperiodic responsiveness.

The EDL server is not reported in the graph since it has basically the same behavior as IPE for almost any load conditions. In particular, simulations showed that for small and medium periodic loads the two algorithms do not have significant differences in their performance. However, even for a high periodic load, the difference is so small that it can be reasonably considered negligible for any practical application.

Although IPE and EDL have very similar performance, they differ significantly in their implementation complexity. As mentioned in previous sections, the EDL algorithm needs to recompute the server parameters quite frequently (namely, when an aperiodic request enters the system and all previous aperiodics have been completely serviced). This overhead can be too expensive in terms of CPU time to use the algorithm in practical applications. On the other hand, in the IPE algorithm the parameters of the server can be computed off-line, and used at run-time to replenish the server capacity.

As shown in the graph, the TBS and IPE algorithms can provide a significant reduction in average aperiodic response time compared to background or polling aperiodic service, whereas the performance of the DPE and DSS algorithms depends on the aperiodic load. For low aperiodic load, DPE and DSS perform as well as TBS and IPE, but as the aperiodic load increases, their performance tends to be similar to that one shown by the Polling Server.
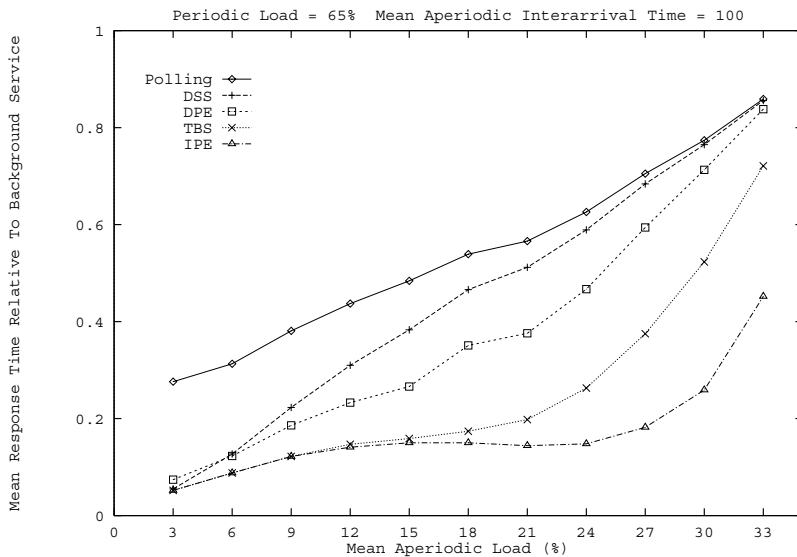
Figure 6.12 Performance of dynamic server algorithms.

Note that in all graphs, TBS and IPE have about the same responsiveness when the aperiodic load is low, and they exhibit a slightly different behavior for heavy aperiodic loads.

All algorithms perform much better when the aperiodic load is generated by a large number of small tasks rather than a small number of long activities. Moreover, note that as the interarrival time T_a increases, and the tasks' execution time becomes longer, the IPE algorithm shows its superiority with respect to the others, which tend to have about the same performance, instead.

The proposed algorithms have been compared with different periodic loads U_p as well. For very low periodic loads all aperiodic service algorithms show a behavior similar to background service. As the periodic load increases, their performance improves substantially with respect to background service. In particular, DPE and DSS have a comparable performance, which tends to approach that of the Polling Server for high periodic loads. On the other hand, TBS and IPE outperform all other algorithms in all situations. The improvement is particularly significant with medium and high workloads. With a very high workload, TBS is no more able to achieve the same good
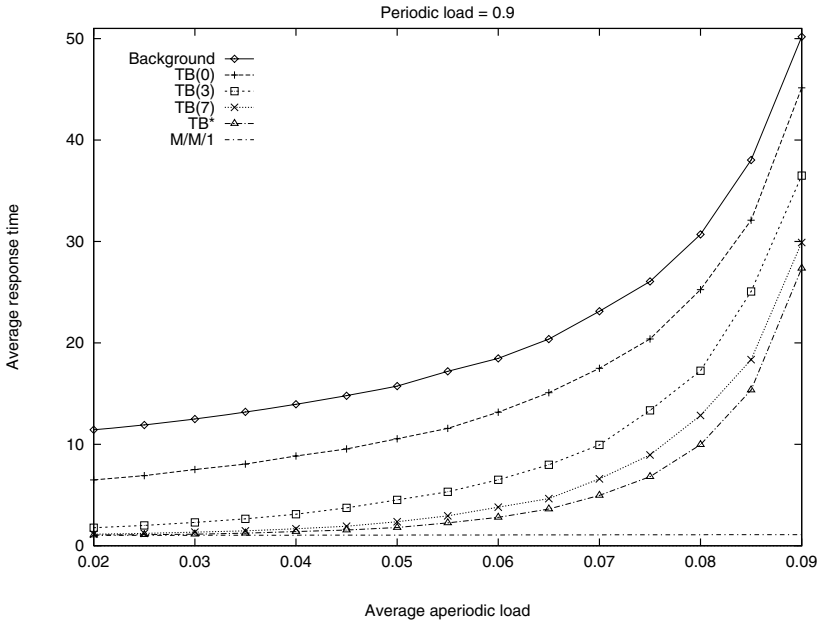
Figure 6.13   Performance results for $U_p = 0.9$.

performance of IPE, even though it is much better than the other algorithms. More extensive simulation results are reported by Spuri and Buttazzo [SB94, SB96].

Simulations have also been conducted to test the performance of the different deadline assignment rules for the Total Bandwidth approach. In Figure 6.13, TB* denotes the optimal algorithm, whereas TB(i) denotes the version of the algorithm that stops iteration after at most i steps from the TBS deadline. Thus, TB(0) coincides with the standard TBS algorithm. In order to show the improvement achieved by the algorithm for each deadline update, the performance of TB* is compared with the one of TB(0), TB(3), and TB(7) that were tested for different periodic and aperiodic loads. To provide a reference term, the response times for background service and for a M/M/1 model are also shown. The plots show the results obtained with a periodic load $U_p = 0.9$. The average response time is plotted with respect to the average task length. Thus, a value of 5 on the y-axis actually means an average response time five times longer than the task computation time.

6.9   THE CONSTANT BANDWIDTH SERVER

In this section we present a novel service mechanism, called the Constant Bandwidth Server (CBS), which efficiently implements a bandwidth reservation strategy. As the DSS, the Constant Bandwidth Server guarantees that, if U_s is the fraction of processor time assigned to a server (i.e., its bandwidth), its contribution to the total utilization factor is no greater than U_s, even in the presence of overloads. Note that this property is not valid for a TBS, whose actual contribution is limited to U_s only under the assumption that all the served jobs execute no more than the declared WCET. With respect to the DSS, however, the CBS shows a much better performance, comparable with the one achievable by a TBS.

The basic idea behind the CBS mechanism can be explained as follows: when a new job enters the system, it is assigned a suitable scheduling deadline (to keep its demand within the reserved bandwidth) and it is inserted in the EDF ready queue. If the job tries to execute more than expected, its deadline is postponed (i.e., its priority is decreased) to reduce the interference on the other tasks. Note that by postponing the deadline, the task remains eligible for execution. In this way, the CBS behaves as a work conserving algorithm, exploiting the available slack in an efficient (deadline-based) way, thus providing better responsiveness with respect to non-work conserving algorithms and to other reservation approaches that schedule the extra portions of jobs in background, as proposed by Mercer, Savage, and Tokuda [MST93, MST94a].

If a subset of tasks is handled by a single server, all the tasks in that subset will share the same bandwidth, so there is no isolation among them. Nevertheless, all the other tasks in the system are protected against overruns occurring in the subset.

In order not to miss any hard deadline, the deadline assignment rules adopted by the server must be carefully designed. The next section precisely defines the CBS algorithm, and formally proves its correctness for any (known or unknown) execution request and arrival pattern.

6.9.1   DEFINITION OF CBS

The CBS can be defined as follows:

- A CBS is characterized by a budget c_s and by an ordered pair (Q_s, T_s), where Q_s is the maximum budget and T_s is the period of the server. The ratio $U_s = Q_s/T_s$ is denoted as the server bandwidth. At each instant, a fixed deadline $d_{s,k}$ is associated with the server. At the beginning $d_{s,0} = 0$.

- Each served job $J_{i,j}$ is assigned a dynamic deadline $d_{i,j}$ equal to the current server deadline $d_{s,k}$.

- Whenever a served job executes, the budget c_s is decreased by the same amount.

- When $c_s = 0$, the server budget is recharged at the maximum value Q_s and a new server deadline is generated as $d_{s,k+1} = d_{s,k} + T_s$. Note that there are no finite intervals of time in which the budget is equal to zero.

- A CBS is said to be active at time t if there are pending jobs (remember the budget c_s is always greater than 0); that is, if there exists a served job $J_{i,j}$ such that $r_{i,j} \leq t < f_{i,j}$. A CBS is said to be idle at time t if it is not active.

- When a job $J_{i,j}$ arrives and the server is active the request is enqueued in a queue of pending jobs according to a given (arbitrary) discipline (e.g., FIFO).

- When a job $J_{i,j}$ arrives and the server is idle, if $c_s \geq (d_{s,k} - r_{i,j})U_s$ the server generates a new deadline $d_{s,k+1} = r_{i,j} + T_s$ and c_s is recharged at the maximum value Q_s, otherwise the job is served with the last server deadline $d_{s,k}$ using the current budget.

- When a job finishes, the next pending job, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle.

- At any instant, a job is assigned the last deadline generated by the server.

6.9.2   SCHEDULING EXAMPLE
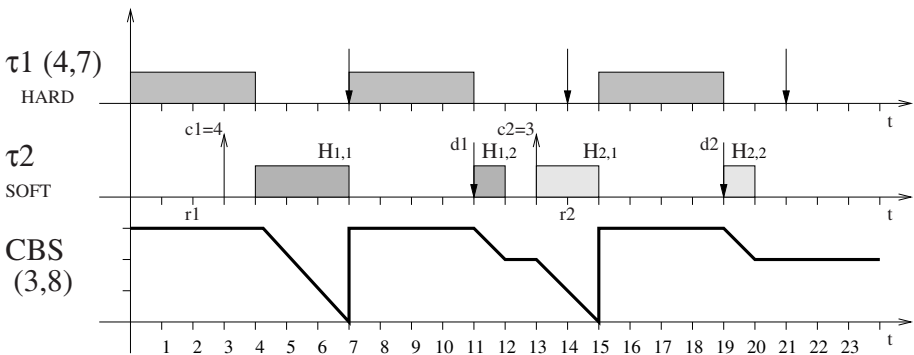
Figure 6.14   An example of CBS scheduling.

Figure 6.14 illustrates an example in which a hard periodic task, τ_1, with computation time $C_1 = 4$ and period $T_1 = 7$, is scheduled together with a soft task, τ_2, served by a CBS having a budget $Q_s = 3$ and a period $T_s = 8$. The first job of τ_2 ($J_{2,1}$), requiring 4 units of execution time, arrives at time $r_1 = 3$, when the server is idle. Being $c_s \geq (d_0 - r_1)U_s$, the job is assigned a deadline $d_1 = r_1 + T_s = 11$ and c_s is recharged at $Q_s = 3$. At time $t = 7$, the budget is exhausted, so a new deadline $d_2 = d_1 + T_s = 19$ is generated and c_s is replenished. Since the server deadline is postponed, τ_1 becomes the task with the earliest deadline and executes until completion. Then, τ_2 resumes and job $J_{2,1}$ (having deadline $d_2 = 19$) is finished at time $t = 12$, leaving a budget $c_s = 2$. The second job of task τ_2 arrives at time $r_2 = 13$ and requires 3 units of time. Since $c_s < (d_2 - r_2)U_s$, the last server deadline d_2 can be used to serve job $J_{2,2}$. At time $t = 15$, the server budget is exhausted, so a new server deadline $d_3 = d_2 + T_s = 27$ is generated and c_s is replenished at Q_s. For this reason, τ_1 becomes the highest priority task and executes until time $t = 19$, when job $J_{1,3}$ finishes and τ_2 can execute, finishing job $J_{2,2}$ at time $t = 20$ leaving a budget $c_s = 2$.

It is worth noting that under a CBS a job J_j is assigned an absolute time-varying deadline d_j that can be postponed if the task requires more than the reserved bandwidth. Thus, each job J_j can be thought as consisting of a number of chunks $H_{j,k}$, each characterized by a release time $a_{j,k}$ and a fixed deadline $d_{j,k}$. An example of chunks produced by a CBS is shown in Figure 6.14. To simplify the notation, we will indicate all the chunks generated by the server with an increasing index k (in the example of Figure 6.14, $H_{1,1} = H_1$, $H_{1,2} = H_2$, $H_{2,1} = H_3$, and so on).

6.9.3 FORMAL DEFINITION

In order to provide a formal definition of the CBS, let a_k and d_k be the release time and the deadline of the kth chunk generated by the server, and let c and n be the actual server budget and the number of pending requests in the server queue (including the request currently being served). These variables are initialized as follows:

$$d_0 = 0, \quad c = 0, \quad n = 0, \quad k = 0.$$

Using this notation, the server behavior can be described by the algorithm shown in Figure 6.15.

```
When job J_j arrives at time r_j
      enqueue the request in the server queue;
      n = n + 1;
      if (n == 1) /* (the server is idle) */
         if (r_j + (c / Q_s) * T_s >= d_k)
            /*--------------Rule 1--------------*/
            k = k + 1;
            a_k = r_j;
            d_k = a_k + T_s;
            c = Q_s;
         else
            /*--------------Rule 2--------------*/
            k = k + 1;
            a_k = r_j;
            d_k = d_{k-1};
            /* c remains unchanged */
When job J_j terminates
      dequeue J_j from the server queue;
      n = n - 1;
      if (n != 0) serve the next job in the queue with deadline d_k;
When job J_j executes for a time unit
      c = c - 1;
When (c == 0)
      /*--------------Rule 3--------------*/
      k = k + 1;
      a_k = actual_time();
      d_k = d_{k-1} + T_s;
      c = Q_s;
```

Figure 6.15   The CBS algorithm.

6.9.4   CBS PROPERTIES

The proposed CBS service mechanism presents some interesting properties that make it suitable for supporting applications with highly variable computation times (e.g., continuous media applications). The most important one, the *isolation property*, is formally expressed by the following theorem and lemma. See the original work by Abeni and Buttazzo [AB98] for the proof.

Theorem 6.8 *The CPU utilization of a CBS S with parameters (Q_s, T_s) is $U_s = \frac{Q_s}{T_s}$, independently from the computation times and the arrival pattern of the served jobs.*

The following lemma provides a simple guarantee test for verifying the feasibility of a task set consisting of hard and soft tasks.

Lemma 6.6 *Given a set of n periodic hard tasks with processor utilization U_p and a set of m CBSs with processor utilization $U_s = \sum_{i=1}^{m} U_{s_i}$, the whole set is schedulable by EDF if and only if*

$$U_p + U_s \leq 1.$$

The isolation property allows us to use a bandwidth reservation strategy to allocate a fraction of the CPU time to soft tasks whose computation time cannot be easily bounded. The most important consequence of this result is that soft tasks can be scheduled together with hard tasks without affecting the a priori guarantee, even in the case in which the execution times of the soft tasks are not known or the soft requests exceed the expected load.

In addition to the isolation property, the CBS has the following characteristics.

- The CBS behaves as a plain EDF algorithm if the served task τ_i has parameters (C_i, T_i) such that $C_i \leq Q_s$ and $T_i = T_s$. This is formally stated by the following lemma.

  Lemma 6.7 *A hard task τ_i with parameters (C_i, T_i) is schedulable by a CBS with parameters $Q_s \geq C_i$ and $T_s = T_i$ if and only if τ_i is schedulable with EDF.*

  Proof. For any job of a hard task we have that $r_{i,j+1} - r_{i,j} \geq T_i$ and $c_{i,j} \leq Q_s$. Hence, by definition of the CBS, each hard job is assigned a deadline $d_{i,j} = r_{i,j} + T_i$ and it is scheduled with a budget $Q_s \geq C_i$. Moreover, since $c_{i,j} \leq Q_s$, each job finishes no later than the budget is exhausted; hence the deadline assigned to a job is never postponed and is exactly the same as the one used by EDF. □

- The CBS automatically reclaims any spare time caused by early completions. This is due to the fact that whenever the budget is exhausted, it is always immediately replenished at its full value and the server deadline is postponed. In this way, the server remains eligible and the budget can be exploited by the pending requests with the current deadline. *This is the main difference with respect to the processor capacity reserves* proposed by Mercer et al. [MST93, MST94a].

- Knowing the statistical distribution of the computation time of a task served by a CBS, it is possible to perform a QoS guarantee based on probabilistic deadlines (expressed in terms of probability for each served job to meet a deadline). Such a statistical analysis is presented by Abeni and Buttazzo [AB98, AB04].

6.9.5   SIMULATION RESULTS

This section shows how the CBS can be efficiently used as a service mechanism for improving responsiveness of soft aperiodic requests. Its performance has been tested against that of TBS and DSS, by measuring the mean tardiness experienced by soft tasks:

$$E_{i,j} \; = \; max\{0, \; f_{i,j} - d_{i,j}\} \tag{6.5}$$

where $f_{i,j}$ is the finishing time of job $J_{i,j}$.

Such a metric was selected because in many soft real-time applications (e.g., multimedia) meeting all soft deadlines is either impossible or very inefficient; hence, the system should be designed to guarantee all the hard tasks and minimize the mean time that soft tasks execute after their deadlines.

All the simulations presented in this section have been conducted on a hybrid task set consisting of 5 periodic hard tasks with fixed parameters and 5 soft tasks with variable execution times and interarrival times. The execution times of the periodic hard tasks were randomly generated in order to achieve a desired processor utilization factor U_{hard}. The execution and interarrival times of the soft tasks were uniformly distributed in order to obtain a mean soft load $\overline{U_{soft}} = \sum_i \frac{c_{i,j}}{r_{i,j+1} - r_{i,j}}$ with $\overline{U_{soft}}$ going from 0 to $1 - U_{hard}$.

The first experiment compares the mean tardiness experienced by soft tasks when they are served by a CBS, a TBS, and a DSS. In this test, the utilization factor of periodic hard tasks was $U_{hard} = 0.5$. The simulation results are illustrated in Figure 6.16, which shows that the performance of the DSS is dramatically worse than the one achieved by the CBS and TBS. The main reason for such different behavior between DSS and CBS is that while the DSS becomes idle until the next replenishing time (that occurs at the server's deadline), the CBS remains eligible by increasing its deadline and replenishing the budget immediately. The TBS does not suffer from this problem; however, *its correct behavior relies on the exact knowledge of WCETs*, so it cannot be used for supporting applications with highly variable computation times.

Figures 6.17 illustrates the results of a similar experiment repeated with $U_{hard} = 0.7$. As we can see, TBS slightly outperforms CBS, but does not protect hard tasks from transient overruns that may occur in the soft activities. Note that since the CBS automatically reclaims any available idle time coming from early completions, for a fair comparison an explicit reclaiming mechanism has also been added in the simulation of the TBS, as described by Spuri, Buttazzo, and Sensini [SBS95].
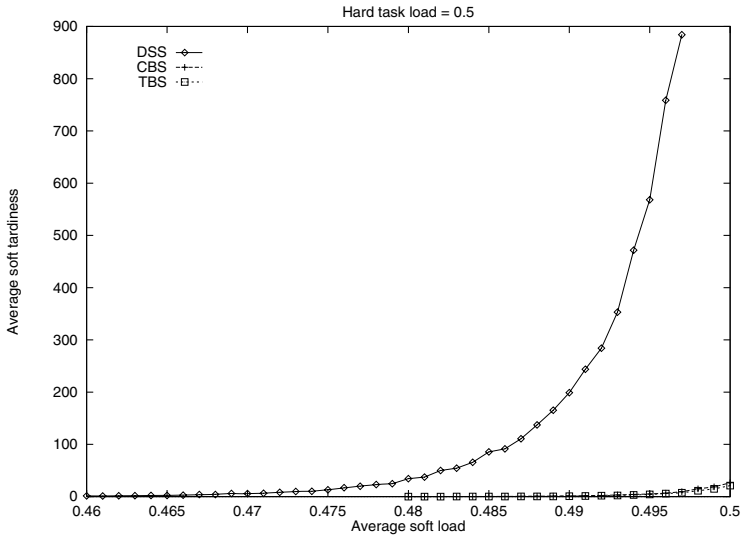
Figure 6.16 First experiment: performance of TBS, CBS and DSS.
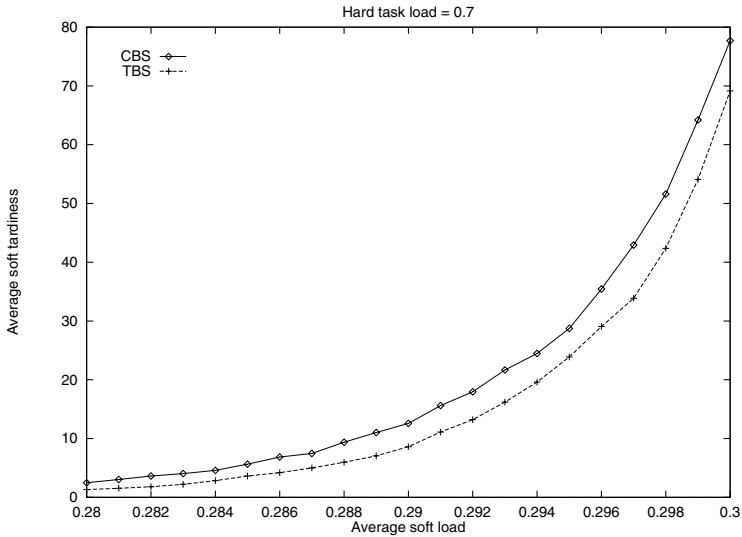
Figure 6.17 Second experiment: CBS against TBS.
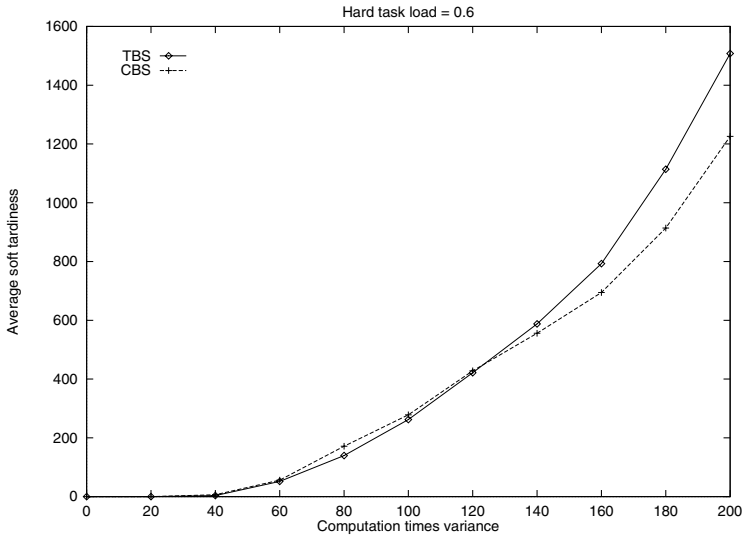
Figure 6.18   Third experiment: CBS against TBS with variable execution times.

The advantage of the CBS over the TBS can be appreciated when $WCET_i >> \overline{c_{i,j}}$. In this case, in fact, the TBS can cause an underutilization of the processor, due to its worst-case assumptions. This fact can be observed in Figure 6.18, which shows the results of a fourth experiment in which $U_{hard} = 0.6$, $\overline{U_{soft}} = 0.4$, the interarrival times are fixed, and the execution times of the soft tasks are uniformly distributed with an increasing variance. As shown in the graph, the CBS performs better than the TBS when tasks' execution times have a high variance. Additional experiments on the CBS are presented in the original work by Abeni and Buttazzo [AB98].

6.9.6   DIMENSIONING CBS PARAMETERS

This section presents a statistical study to evaluate the effects of the CBS parameters (Q_s, T_s) on task response times, and proposes a technique to compute the parameters that minimize the average response time of the served tasks [BB06].

The worst-case response time R_i of a job with computation time C_i served by a CBS with bandwidth U_s is a function of the server budget Q_s. For the sake of clarity, R_i is first derived by neglecting the overhead, and then modified to take the overhead into account.

From the CBS analysis, we know that, if the task set is feasible, that is, if the total processor utilization is less than 1, then the served job can never miss the current server deadline. Hence, the maximum response time R_i occurs when the other tasks in the system create the maximum interference on the server. If the computation time C_i of the served job is exactly a multiple of the server budget Q_s, then the job finishes at the server deadline; that is,

$$R_i = \frac{C_i}{Q_s}T_s = \frac{C_i}{U_s}. \tag{6.6}$$

More generally, if the computation time C_i of the job is not multiple of the budget Q_s, the last portion of the job will not finish at the server deadline, but it will finish at most Δ_i units before the deadline, as shown in Figure 6.19, where

$$\Delta_i = \left\lceil \frac{C_i}{Q_s} \right\rceil Q_s - C_i. \tag{6.7}$$
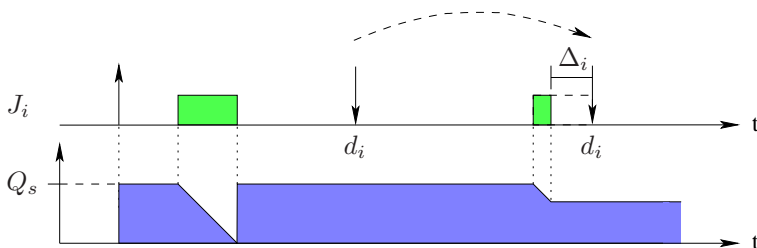
Figure 6.19   Worst-case finishing time of a job served by a CBS.

Hence, the response time of the job becomes

$$
\begin{aligned}
R_i &= \left\lceil \frac{C_i}{Q_s} \right\rceil T_s - \Delta_i \\
&= \left\lceil \frac{C_i}{Q_s} \right\rceil T_s - \left(\left\lceil \frac{C_i}{Q_s} \right\rceil Q_s - C_i \right) \\
&= C_i + \left\lceil \frac{C_i}{Q_s} \right\rceil (T_s - Q_s). \tag{6.8}
\end{aligned}
$$

Figure 6.20 illustrates the worst-case response time of a CBS as a function of the budget.
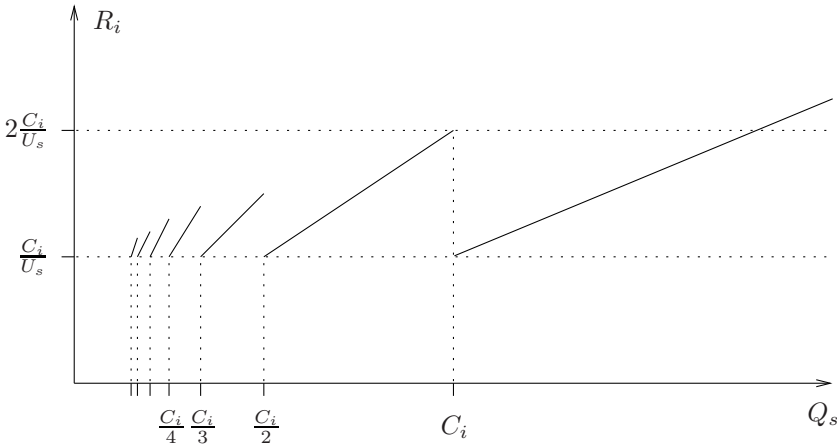
Figure 6.20   Worst-case response time of a CBS as a function of the budget.

From the graph shown in Figure 6.20 it is clear that for a given job with constant execution time C_i, the minimum worst-case response time is C_i/U_s and can be achieved when C_i is a perfect multiple of Q_s. In practice, however, task execution time varies, inducing response time fluctuations due to the bandwidth enforcement mechanism achieved through deadline postponements. From Figure 6.20 it is also clear that such fluctuations would be reduced by making the budget very small compared to the average execution time, so that the server would approximate the ideal fluid server. Unfortunately, however, a small budget (which means a short server period) causes the job to be split in many small chunks, increasing the runtime overhead. As a consequence, to properly set the server granularity T_s, the runtime overhead must be taken into account in the analysis.

TAKING OVERHEADS INTO ACCOUNT

Whenever the budget is exhausted, the server deadline is postponed, so the served job can be preempted by other tasks with earliest deadline. If ϵ denotes the time needed for a context switch, then the overhead introduced by the CBS can be taken into account by subtracting such a time from the server budget. Hence, Equation (6.8) can be modified as follows:

$$
\begin{aligned}
R_i &= C_i + \left\lceil \frac{C_i}{Q_s - \epsilon} \right\rceil (T_s - Q_s + \epsilon) \\
&= C_i + \left\lceil \frac{C_i}{T_s U_s - \epsilon} \right\rceil (T_s - T_s U_s + \epsilon).
\end{aligned}
\tag{6.9}
$$
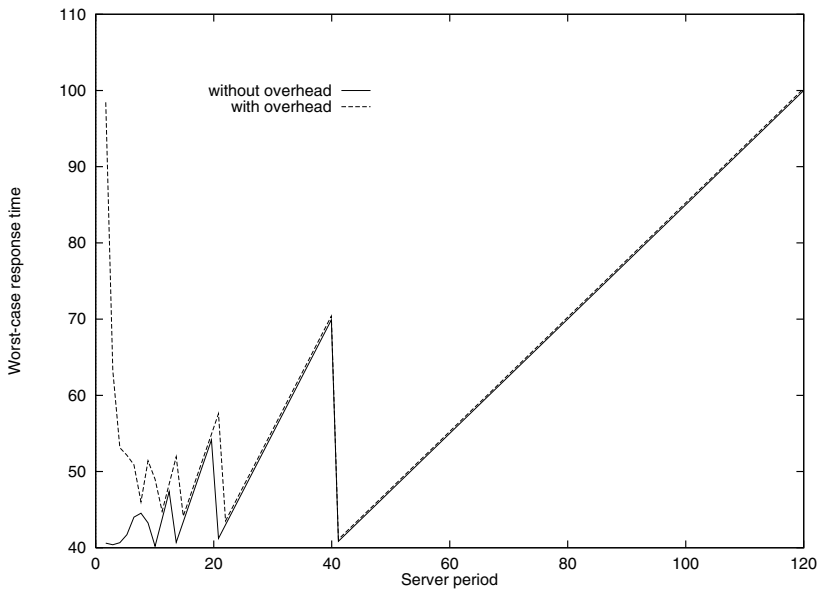
Figure 6.21 Worst-case response time of a CBS as a function of the period.

Figure 6.21 illustrates the worst-case response time of a CBS as a function of the period, with and without overhead. Equation (6.9) has been plotted for $C_i = 10$, $U_s = 0.25$, and $\epsilon = 0.2$. As is clear from the plot, the overhead prevents using small values of the period; hence, it is interesting to find the value of the server period that minimizes the response time. Note that for computing the probability distribution function of the response time, we actually need to express the response time as a function of the job execution time C_i.

Figure 6.22 illustrates the worst-case response time of a CBS as a function of the job execution time. As shown in Figure 6.22, the response time R_i can be upper bounded by the following linear function

$$R_i^{\text{ub}} = T_s - Q_s + \epsilon + \frac{T_s}{Q_s - \epsilon} C_i \tag{6.10}$$

and lower bounded by

$$R_i^{\text{lb}} = \frac{T_s}{Q_s - \epsilon} C_i. \tag{6.11}$$
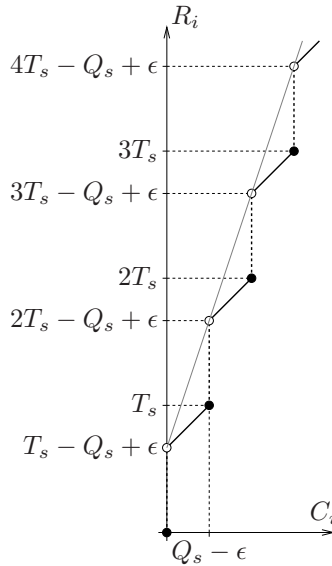
Figure 6.22   Worst-case response time of a CBS as a function of the job execution time.

We now consider the problem of selecting the best CBS parameters, such that the average task response time R_i is minimized. For this purpose we suppose to have the probability density function (p.d.f.) $f_C(c)$ of the task execution time, and the respective cumulative distribution function (c.d.f.) $F_C(c)$, representing the probability that the execution time is smaller than or equal to c. That is,

$$F_C(c) = \int_0^c f_C(x)\mathrm{d}x. \tag{6.12}$$

Since the minimization of the average R_i can in general be too complex, we consider the problem of minimizing its linear upper bound R_i^{ub}. In this case, the average response time R_i^{avg} is computed as follows:

$$
\begin{aligned}
R_i^{\mathrm{avg}} &= \int_0^{+\infty} \left(T_s - Q_s + \epsilon + \frac{T_s}{Q_s - \epsilon}x \right) f_C(x)\mathrm{d}x \\
&= T_s - Q_s + \epsilon + \frac{T_s}{Q_s - \epsilon}C^{\mathrm{avg}} \\
&= T_s(1 - U_s) + \epsilon + \frac{T_s}{T_sU_s - \epsilon}C^{\mathrm{avg}} \tag{6.13}
\end{aligned}
$$

Hence, the period T_s that minimizes the average response time R_i^{avg} can be computed by simple functional analysis. Thus, we have

$$\frac{\mathrm{d}R_i^{\text{avg}}}{\mathrm{d}T_s} = 1 - U_s - \frac{\epsilon}{(T_s U_s - \epsilon)^2} C^{\text{avg}}, \tag{6.14}$$

which is equal to zero when

$$T_s = \frac{1}{U_s} \left(\epsilon + \sqrt{\frac{\epsilon C^{\text{avg}}}{1 - U_s}} \right). \tag{6.15}$$

6.10 SUMMARY

The experimental simulations have established that from a performance point of view, IPE, EDL, and TB* show the best results for reducing aperiodic responsiveness. Although optimal, however, EDL is far from being practical, due to the overall complexity. On the other hand, IPE and TB* achieve a comparable performance with much less computational overhead. Moreover, both EDL and IPE may require significant memory space when task periods are not harmonically related.

The Total Bandwidth algorithm also shows a good performance, sometimes comparable to that of the nearly optimal IPE. Observing that its implementation complexity is among the simplest, the TBS algorithm could be a good candidate for practical systems. In addition, the TBS deadline assignment rule can be tuned to enhance aperiodic responsiveness up to the optimal TB* behavior. Compared to IPE and EDL, TB* does not require large memory space, and the optimal deadline can be computed in $O(Nn)$ complexity, N being the maximum number of steps that have to be done for each task to shorten its initial deadline (assigned by the TBS rule). As for the EDL server, this is a pseudo-polynomial complexity since in the worst case N can be large.

One major problem of the TBS and TB* algorithms is that they do not use a server budget for controlling aperiodic execution, but rely on the knowledge of the worst-case computation time specified by each job at its arrival. When such a knowledge is not available, not reliable, or too pessimistic (due to highly variable execution times), then hard tasks are not protected from transient overruns occurring in the soft tasks and could miss their deadlines. The CBS algorithm can be efficiently used in these situations, since it has a performance comparable to the one of the TBS and also provides temporal isolation, by limiting the bandwidth requirements of the served tasks to the value U_s specified at design time.

Figure 6.23   Evaluation summary of dynamic-priority servers.

Figure 6.23 provides a qualitative evaluation of the algorithms presented in this chapter in terms of performance, computational complexity, memory requirement, and implementation complexity.

Exercises

6.1   Compute the maximum processor utilization that can be assigned to a Dynamic Sporadic Server to guarantee the following periodic tasks, under EDF:

	C_i	T_i
τ_1	2	6
τ_2	3	9

6.2 Together with the periodic tasks illustrated in Exercise 6.1, schedule the following aperiodic tasks with a Dynamic Sporadic Server with $C_s = 2$ and $T_s = 6$.

	a_i	C_i
J_1	1	3
J_2	5	1
J_3	15	1

6.3 Solve the same scheduling problem described in Exercise 6.2 with a Total Bandwidth Server having utilization $U_s = 1/3$.

6.4 Solve the same scheduling problem described in Exercise 6.2 with a Constant Bandwidth Server with $C_s = 2$ and $T_s = 6$.

6.5 Solve the same scheduling problem described in Exercise 6.2 with an Improved Total Bandwidth Server with $U_s = 1/3$, which performs only one shortening step.

6.6 Solve the same scheduling problem described in Exercise 6.2 with the optimal Total Bandwidth Server (TB*).

6.7 Consider the following set of periodic tasks:

	C_i	T_i
τ_1	4	10
τ_2	4	12

After defining two Total Bandwidth Servers, TB_1 and TB_2, with utilization factors $U_{s1} = 1/10$ and $U_{s2} = 1/6$, construct the EDF schedule in the case in which two aperiodic requests $J_1(a_1 = 1, C_1 = 1)$ and $J_2(a_2 = 9, C_2 = 1)$ are served by TB_1, and two aperiodic requests $J_3(a_3 = 2, C_3 = 1)$ and $J_4(a_4 = 6, C_4 = 2)$ are served by TB_2.

6.8 A control application consists of two periodic tasks with computation times $C_1 = 8$, $C_2 = 6$ ms, and periods $T_1 = 20$, $T_2 = 30$ ms. Moreover, the system includes two interrupt handling routines, with computation times of 1.0 and 1.4 ms each. Considering a context switch cost of 20 μs, compute the CBS parameters that minimize the average response time of the interrupts.