
FIXED-PRIORITY SERVERS

5.1 INTRODUCTION

The scheduling algorithms treated in the previous chapters deal with homogeneous sets of tasks, where all computational activities are either aperiodic or periodic. Many real-time control applications, however, require both types of processes, which may also differ for their criticalness. Typically, periodic tasks are time-driven and execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates. Aperiodic tasks are usually event-driven and may have hard, soft, or non-real-time requirements depending on the specific application.

When dealing with hybrid task sets, the main objective of the kernel is to guarantee the schedulability of all critical tasks in worst-case conditions and provide good average response times for soft and non-real-time activities. Off-line guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment; that is, by assuming a maximum arrival rate for each critical event. This implies that aperiodic tasks associated with critical events are characterized by a minimum interarrival time between consecutive instances, which bounds the aperiodic load. Aperiodic tasks characterized by a minimum interarrival time are called *sporadic*. They are guaranteed under peak-load situations by assuming their maximum arrival rate.

If the maximum arrival rate of some event cannot be bounded a priori, the associated aperiodic task cannot be guaranteed off-line, although an on-line guarantee of individual aperiodic requests can still be done. Aperiodic tasks requiring on-line guarantee on individual instances are called *firm*. Whenever

a firm aperiodic request enters the system, an acceptance test can be executed by the kernel to verify whether the request can be served within its deadline. If such a guarantee cannot be done, the request is rejected.

In the next sections, we present a number of scheduling algorithms for handling hybrid task sets consisting of a subset of hard periodic tasks and a subset of soft aperiodic tasks. All algorithms presented in this chapter rely on the following assumptions:

- Periodic tasks are scheduled based on a fixed-priority assignment; namely, the Rate-Monotonic (RM) algorithm;
- All periodic tasks start simultaneously at time $t = 0$ and their relative deadlines are equal to their periods.
- Arrival times of aperiodic requests are unknown.
- When not explicitly specified, the minimum interarrival time of a sporadic task is assumed to be equal to its deadline.

Aperiodic scheduling under dynamic priority assignment is discussed in the next chapter.

5.2 BACKGROUND SCHEDULING

The simplest method to handle a set of soft aperiodic activities in the presence of periodic tasks is to schedule them in background; that is, when there are not periodic instances ready to execute. The major problem with this technique is that, for high periodic loads, the response time of aperiodic requests can be too long for certain applications. For this reason, background scheduling can be adopted only when the aperiodic activities do not have stringent timing constraints and the periodic load is not high.

Figure 5.1 illustrates an example in which two periodic tasks are scheduled by RM, while two aperiodic tasks are executed in background. Since the processor utilization factor of the periodic task set ($U = 0.73$) is less than the least upper bound for two tasks ($U_{lub}(2) \simeq 0.83$), the periodic tasks are schedulable by RM. Notice that the guarantee test does not change in the presence of aperiodic requests, since background scheduling does not influence the execution of periodic tasks.

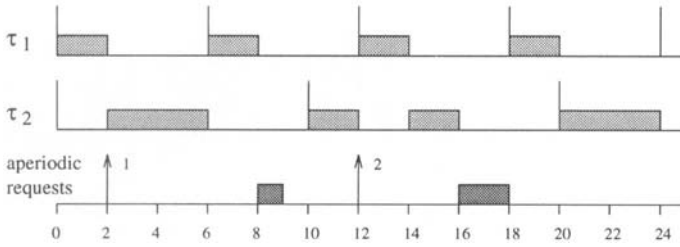


Figure 5.1 Example of background scheduling of aperiodic requests under Rate Monotonic.

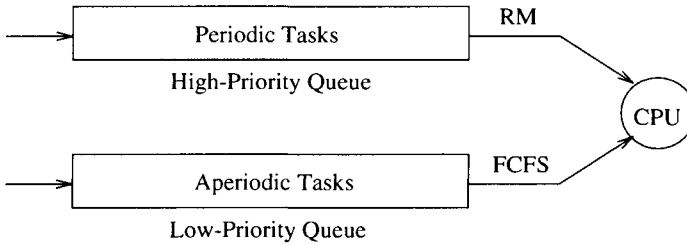


Figure 5.2 Scheduling queues required for background scheduling.

The major advantage of background scheduling is its simplicity. As shown in Figure 5.2, two queues are needed to implement the scheduling mechanism: one (with a higher priority) dedicated to periodic tasks and the other (with a lower priority) reserved for aperiodic requests. The two queueing strategies are independent and can be realized by different algorithms, such as RM for periodic tasks and First Come First Served (FCFS) for aperiodic requests. Tasks are taken from the aperiodic queue only when the periodic queue is empty. The activation of a new periodic instance causes any aperiodic tasks to be immediately preempted.

5.3 POLLING SERVER

The average response time of aperiodic tasks can be improved with respect to background scheduling through the use of a *server*; that is, a periodic task whose purpose is to service aperiodic requests as soon as possible. Like any periodic task, a server is characterized by a period T_s and a computation time C_s , called

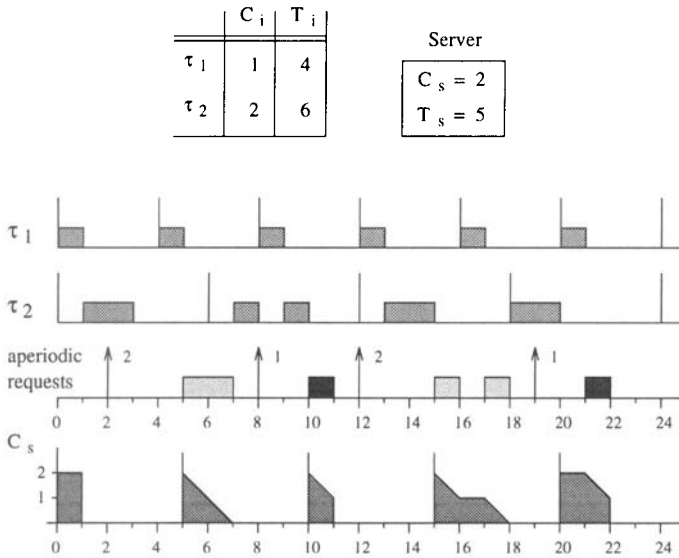


Figure 5.3 Example of a Polling Server scheduled by RM.

server *capacity*. In general, the server is scheduled with the same algorithm used for the periodic tasks, and, once active, it serves the aperiodic requests within the limit of its server capacity. The ordering of aperiodic requests does not depend on the scheduling algorithm used for periodic tasks, and it can be done by arrival time, computation time, deadline, or any other parameter.

The *Polling Server* (PS) is an algorithm based on such an approach. At regular intervals equal to the period T_s , PS becomes active and serves any pending aperiodic requests within the limit of its capacity C_s . If no aperiodic requests are pending, PS suspends itself until the beginning of its next period, and the time originally allocated for aperiodic service is not preserved for aperiodic execution but is used by periodic tasks [LSS87, SSL89]. Note that if an aperiodic request arrives just after the server has suspended, it must wait until the beginning of the next polling period, when the server capacity is replenished at its full value.

Figure 5.3 illustrates an example of aperiodic service obtained through a Polling Server scheduled by RM. The aperiodic requests are reported on the third row, whereas the fourth row shows the server capacity as a function of time. Numbers beside the arrows indicate the computation times associated with the requests.

In the example shown in Figure 5.3, the Polling Server has a period $T_s = 5$ and a capacity $C_s = 2$, so it runs with an intermediate priority with respect to the other periodic tasks. At time $t = 0$, the processor is assigned to task τ_1 , which is the highest-priority task according to RM. At time $t = 1$, τ_1 completes its execution and the processor is assigned to PS. However, since no aperiodic requests are pending, the server suspends itself and its capacity is used by periodic tasks. As a consequence, the request arriving at time $t = 2$ cannot receive immediate service but must wait until the beginning of the second server period ($t = 5$). At this time, the capacity is replenished at its full value ($C_s = 2$) and used to serve the aperiodic task until completion. Note that, since the capacity has been totally consumed, no other aperiodic requests can be served in this period; thus, the server becomes idle.

The second aperiodic request receives the same treatment. However, note that since the second request only uses half of the server capacity, the remaining half is discarded because no other aperiodic tasks are pending. Also note that, at time $t = 16$, the third aperiodic request is preempted by task τ_1 , and the server capacity is preserved.

5.3.1 Schedulability analysis

We first consider the problem of guaranteeing a set of hard periodic tasks in the presence of soft aperiodic tasks handled by a Polling Server. Then we show how to derive a schedulability test for firm aperiodic requests.

The schedulability of periodic tasks can be guaranteed by evaluating the interference introduced by the Polling Server on periodic execution. In the worst case, such an interference is the same as the one introduced by an equivalent periodic task having a period equal to T_s and a computation time equal to C_s . In fact, independently of the number of aperiodic tasks handled by the server, a maximum time equal to C_s is dedicated to aperiodic requests at each server period. As a consequence, the processor utilization factor of the Polling Server is $U_s = C_s/T_s$, and hence the schedulability of a periodic set with n tasks and utilization U_p can be guaranteed if

$$U_p + U_s \leq U_{lub}(n + 1).$$

If periodic tasks (including the server) are scheduled by RM, the schedulability test becomes

$$\sum_{i=1}^n \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq (n + 1)[2^{1/(n+1)} - 1].$$

A more precise schedulability test for aperiodic servers that behave like a periodic task will be derived in Section 5.5 for the Priority Exchange algorithm. Note that more Polling Servers can be created and execute concurrently on different aperiodic task sets. For example, a high-priority server could be reserved for a subset of important aperiodic tasks, whereas a lower-priority server could be used to handle less important requests. In general, in the presence of m servers, a set of n periodic tasks is guaranteed if

$$U_p + \sum_{j=1}^m U_{s_j} \leq U_{lub}(n + m).$$

5.3.2 Aperiodic guarantee

In order to analyze the schedulability of firm aperiodic activities under a Polling Server, consider the case of a single aperiodic request J_a , arrived at time r_a , with computation time C_a and deadline D_a . Since an aperiodic request can wait for at most one period before receiving service, if $C_a \leq C_s$ the request is certainly completed within two server periods. Thus, it is guaranteed if

$$2T_s \leq D_a.$$

For arbitrary computation times, the aperiodic request is certainly completed in $\lceil C_a/C_s \rceil$ server periods; hence, it is guaranteed if

$$T_s + \left\lceil \frac{C_a}{C_s} \right\rceil T_s \leq D_a.$$

This schedulability test is only sufficient because it does not consider when the server executes within its period. A sufficient and necessary schedulability test can be found for the case in which PS has the highest priority among the periodic tasks; that is, the shortest period. In this case, in fact, it always executes at the beginning of its periods, so that the finishing time of the aperiodic request can be estimated precisely. As shown in Figure 5.4, by defining

$$\begin{aligned} F_a &= \left\lfloor \frac{C_a}{C_s} \right\rfloor \\ G_a &= \left\lceil \frac{r_a}{T_s} \right\rceil \end{aligned}$$

the initial delay of request J_a is given by $(G_a T_s - r_a)$. Then, since $F_a C_s$ is the total capacity consumed by J_a in F_a server periods, the residual execution to be done in the next server period is

$$R_a = C_a - F_a C_s.$$

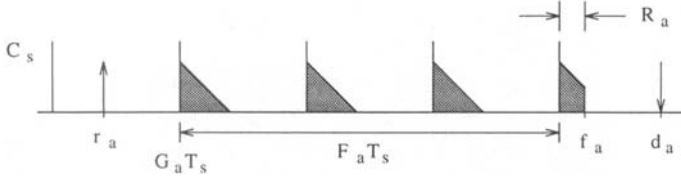


Figure 5.4 Calculation of the finishing time of an aperiodic request scheduled by a Polling Server having the highest priority.

As a consequence, the aperiodic finishing time can be computed as

$$f_a = G_a T_s + F_a T_s + R_a,$$

and its schedulability can be guaranteed if and only if $f_a \leq d_a$, being d_a the absolute deadline of the request ($d_a = r_a + D_a$). Thus, the resulting schedulability condition is

$$(F_a + G_a)T_s + R_a \leq d_a.$$

This result can be extended to a set of firm aperiodic requests ordered in a queue by increasing deadline. In this case, at any time t , the total aperiodic computation that has to be served in any interval $[t, d_k]$ is equal to the sum of the remaining processing times $c_i(t)$ of the tasks with deadline $d_i \leq d_k$; that is,

$$C_{ape}(t, d_k) = \sum_{i=1}^k c_i(t).$$

Note that, if $c_s(t)$ is the residual server capacity at time t and PS has the highest priority, a portion of C_{ape} equal to $c_s(t)$ is immediately executed in the current period. Hence, the finishing time of request J_k can be computed as

$$f_k = \begin{cases} t + C_{ape}(t, d_k) & \text{if } C_{ape}(t, d_k) \leq c_s(t) \\ (F_k + G_k)T_s + R_k & \text{otherwise,} \end{cases}$$

where

$$\begin{cases} F_k = \left\lfloor \frac{C_{ape}(t, d_k) - c_s(t)}{C_s} \right\rfloor \\ G_k = \left\lceil \frac{t}{T_s} \right\rceil \\ R_k = C_{ape}(t, d_k) - c_s(t) - F_k C_s. \end{cases}$$

Once all finishing times have been calculated, the set of firm aperiodic requests is guaranteed at time t if and only if

$$f_k \leq d_k \quad \forall k = 1, \dots, n.$$

5.4 DEFERRABLE SERVER

The *Deferrable Server* (DS) algorithm is a service technique introduced by Lehoczky, Sha, and Strosnider in [LSS87, SLS95] to improve the average response time of aperiodic requests with respect to polling service. As the Polling Server, the DS algorithm creates a periodic task (usually having a high priority) for servicing aperiodic requests. However, unlike polling, DS preserves its capacity if no requests are pending upon the invocation of the server. The capacity is maintained until the end of the period, so that aperiodic requests can be serviced at the same server's priority at anytime, as long as the capacity has not been exhausted. At the beginning of any server period, the capacity is replenished at its full value.

The DS algorithm is illustrated in Figure 5.5 using the same task set and the same server parameters ($C_s = 2$, $T_s = 5$) considered in Figure 5.3. At time $t = 1$, when τ_1 is completed, no aperiodic requests are pending; hence, the processor is assigned to task τ_2 . However, the DS capacity is not used for periodic tasks, but it is preserved for future aperiodic arrivals. Thus, when the first aperiodic request arrives at time $t = 2$, it receives immediate service. Since the capacity of the server is exhausted at time $t = 4$, no other requests can be serviced before the next period. At time $t = 5$, C_s is replenished at its full value and preserved until the next arrival. The second request arrives at time $t = 8$, but it is not served immediately because τ_1 is active and has a higher priority.

Thus, DS provides much better aperiodic responsiveness than polling, since it preserves the capacity until it is needed. Shorter response times can be achieved by creating a Deferrable Server having the highest priority among the periodic tasks. An example of high-priority DS is illustrated in Figure 5.6. Notice that the second aperiodic request preempts task τ_1 , being $C_s > 0$ and $T_s < T_1$, and it entirely consumes the capacity at time $t = 10$. When the third request arrives at time $t = 11$, the capacity is zero; hence, its service is delayed until the beginning of the next server period. The fourth request receives the same treatment because it arrives at time $t = 16$, when C_s is exhausted.

5.4.1 Schedulability analysis

Any schedulability analysis related to the Rate-Monotonic algorithm has been done on the implicit assumption that a periodic task must execute whenever it is the highest-priority task ready to run. It is easy to see that the De-

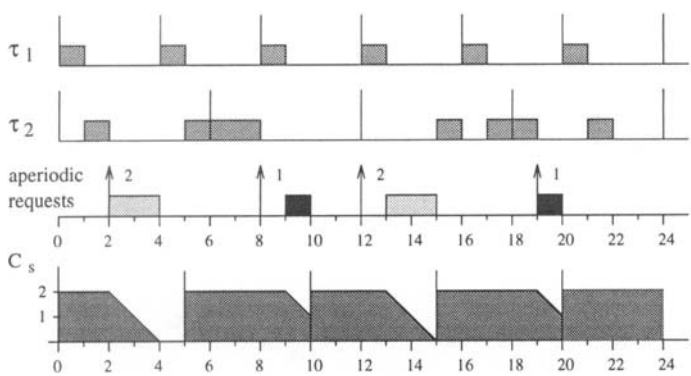


Figure 5.5 Example of a Deferrable Server scheduled by RM.

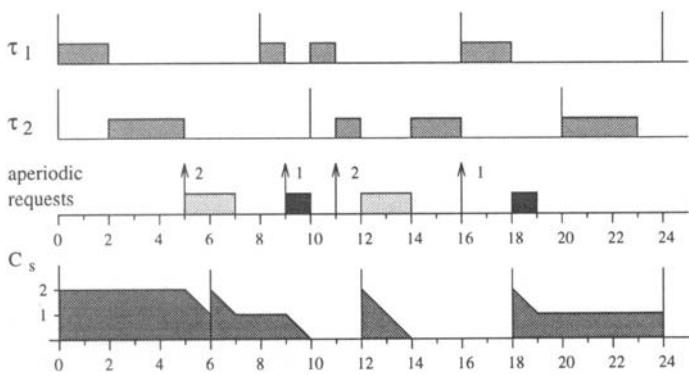


Figure 5.6 Example of high-priority Deferrable Server.

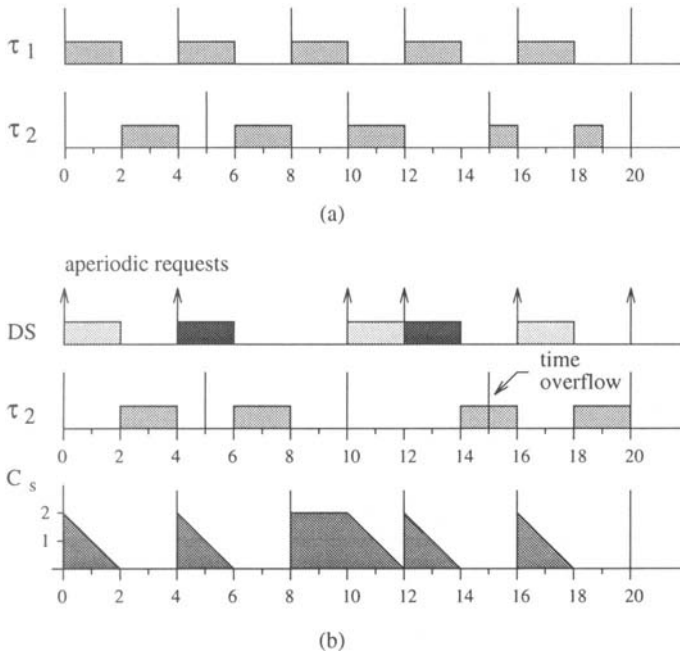


Figure 5.7 DS is not equivalent to a periodic task. In fact, the periodic set $\{\tau_1, \tau_2\}$ is schedulable by RM (a); however, if we replace τ_1 with DS, τ_2 misses its deadline (b).

ferrable Server violates this basic assumption. In fact, the schedule illustrated in Figure 5.6 shows that DS does not execute at time $t = 0$, although it is the highest-priority task ready to run, but it *defers* its execution until time $t = 5$, which is the arrival time of the first aperiodic request.

If a periodic task *defers* its execution when it could execute immediately, then a lower-priority task could miss its deadline even if the task set was schedulable. Figure 5.7 illustrates this phenomenon by comparing the execution of a periodic task to the one of a Deferrable Server with the same period and execution time.

The periodic task set considered in this example consists of two tasks, τ_1 and τ_2 , having the same computation time ($C_1 = C_2 = 2$) and different periods ($T_1 = 4$, $T_2 = 5$). As shown in Figure 5.7a, the two tasks are schedulable by RM. However, if τ_1 is replaced with a Deferrable Server having the same period and execution time, the low-priority task τ_2 can miss its deadline depending on the sequence of aperiodic arrivals. Figure 5.7b shows a particular sequence

of aperiodic requests that cause τ_2 to miss its deadline at time $t = 15$. This happens because, at time $t = 8$, DS does not execute (as a normal periodic task would do) but preserves its capacity for future requests. This deferred execution, followed by the servicing of two consecutive aperiodic requests in the interval $[10, 14]$, prevents task τ_2 from executing during this interval, causing its deadline to be missed.

Such an invasive behavior of the Deferrable Server results in a lower schedulability bound for the periodic task set. The calculation of the least upper bound of the processor utilization factor in the presence of Deferrable Server is shown in the next section.

Calculation of U_{lub} for $RM+DS$

The schedulability bound for a set of periodic tasks with a Deferrable Server is derived under the same basic assumptions used in Chapter 4 to compute U_{lub} for RM. To simplify the computation of the bound for n tasks, we first determine the worst-case relations among the tasks, and then we derive the lower bound against the worst-case model [LSS87].

Consider a set of n periodic tasks, τ_1, \dots, τ_n , ordered by increasing periods, and a Deferrable Server with a higher priority. The worst-case condition for the periodic tasks, as derived for the RM analysis, is such that $T_1 < T_n < 2T_1$. In the presence of a DS, however, the derivation of the worst-case is more complex and requires the analysis of three different cases, as discussed in [SLS95]. For the sake of clarity, here we analyze one case only, the most general, in which DS may execute three times within the period of the highest-priority periodic task. This happens when DS defers its service at the end of its period and also executes at the beginning of the next period. In this situation, depicted in Figure 5.8, the full processor utilization is achieved by the following tasks' parameters:

$$\begin{cases} C_s = T_1 - (T_s + C_s) = \frac{T_1 - T_s}{2} \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \dots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_s - C_s - \sum_{i=1}^{n-1} C_i = \frac{3T_s + T_1 - 2T_n}{2}. \end{cases}$$

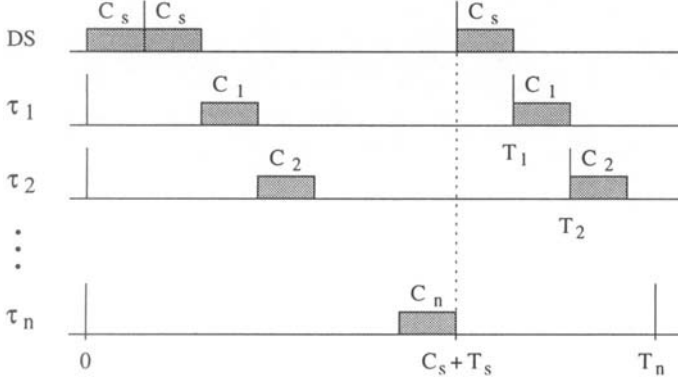


Figure 5.8 Worst-case task relations for a Deferrable Server.

Hence, the resulting utilization is

$$\begin{aligned}
 U &= \frac{C_s}{T_s} + \frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} = \\
 &= U_s + \frac{T_2 - T_1}{T_1} + \dots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{3T_s + T_1 - 2T_n}{2T_n} = \\
 &= U_s + \frac{T_2}{T_1} + \dots + \frac{T_n}{T_{n-1}} + \left(\frac{3T_s}{2T_1} + \frac{1}{2}\right) \frac{T_1}{T_n} - n.
 \end{aligned}$$

Defining

$$\begin{cases} R_s = T_1/T_s \\ R_i = T_{i+1}/T_i \\ K = \frac{1}{2}(3T_s/T_1 + 1) \end{cases}$$

and noting that

$$R_1 R_2 \dots R_{n-1} = \frac{T_n}{T_1},$$

the utilization factor may be written as

$$U = U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{R_1 R_2 \dots R_{n-1}} - n.$$

Following the approach used for RM, we minimize U over R_i , $i = 1, \dots, n-1$.

Hence,

$$\frac{\partial U}{\partial R_i} = 1 - \frac{K}{R_i^2 (\prod_{j \neq i}^{n-1} R_j)}.$$

Thus, defining $P = R_1 R_2 \dots R_{n-1}$, U is minimum when

$$\begin{cases} R_1 P = K \\ R_2 P = K \\ \dots \\ R_{n-1} P = K \end{cases}$$

that is, when all R_i have the same value:

$$R_1 = R_2 = \dots = R_{n-1} = K^{1/n}.$$

Substituting this value in U we obtain

$$\begin{aligned} U_{lub} - U_s &= (n-1)K^{1/n} + \frac{K}{K^{(1-1/n)}} - n = \\ &= nK^{1/n} - K^{1/n} + K^{1/n} - n = \\ &= n(K^{1/n} - 1) \end{aligned}$$

that is,

$$U_{lub} = U_s + n(K^{1/n} - 1). \quad (5.1)$$

Now, noting that

$$U_s = \frac{C_s}{T_s} = \frac{T_1 - T_s}{2T_s} = \frac{R_s - 1}{2}$$

we have

$$R_s = (2U_s + 1).$$

Thus, K can be rewritten as

$$K = \left(\frac{3}{2R_s} + \frac{1}{2} \right) = \frac{U_s + 2}{2U_s + 1},$$

and finally

$$U_{lub} = U_s + n \left[\left(\frac{U_s + 2}{2U_s + 1} \right)^{1/n} - 1 \right]. \quad (5.2)$$

Taking the limit as $n \rightarrow \infty$, we find the worst-case bound as a function of U_s to be given by

$$\lim_{n \rightarrow \infty} U_{lub} = U_s + \ln\left(\frac{U_s + 2}{2U_s + 1}\right). \quad (5.3)$$

Thus, given a set of n periodic tasks and a Deferrable Server with utilization factors U_p and U_s , respectively, the schedulability of the periodic task set is guaranteed under RM if

$$U_p + U_s \leq U_{lub}$$

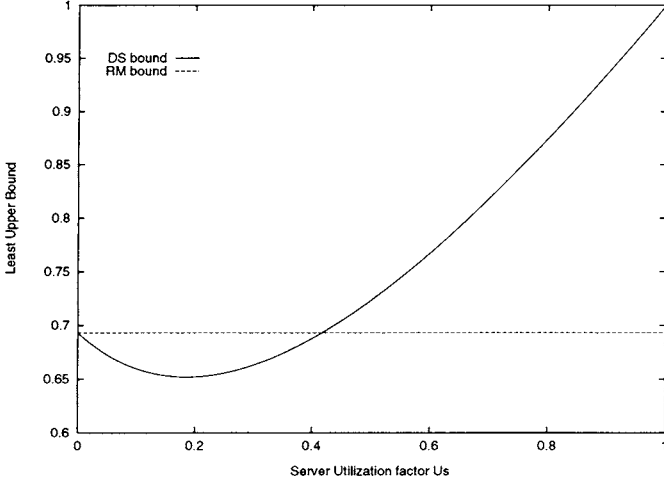


Figure 5.9 Schedulability bound for periodic tasks and DS as a function of the server utilization factor U_s .

that is, if

$$U_p \leq \ln\left(\frac{U_s + 2}{2U_s + 1}\right). \quad (5.4)$$

A plot of equation (5.3) as a function of U_s is shown in Figure 5.9. For comparison, the RM bound is also reported in the plot. Notice that for $U_s < 0.4$ the presence of DS worsens the RM bound, whereas for $U_s > 0.4$ the RM bound is improved.

Deriving equation (5.3) with respect to U_s , we can find the absolute minimum value of U_{lub} :

$$\frac{\partial U_{lub}}{\partial U_s} = 1 + \frac{(2U_s + 1)}{(U_s + 2)} \frac{(2U_s + 1) - 2(U_s + 2)}{(2U_s + 1)^2} = \frac{2U_s^2 + 5U_s - 1}{(U_s + 2)(2U_s + 1)}.$$

The value of U_s that minimizes the above expression is

$$U_s^* = \frac{\sqrt{33} - 5}{4} \simeq 0.186,$$

so the minimum value of U_{lub} is $U_{lub}^* \simeq 0.652$.

5.4.2 Aperiodic guarantee

The schedulability analysis for firm aperiodic tasks is derived by assuming a high-priority Deferrable Server. A guarantee test for a single request is first derived and then extended to a set of aperiodic tasks. Since DS preserves its execution time, let $c_s(t)$ be the value of its capacity at time t . Then, when a firm aperiodic request $J_a(C_a, D_a)$ enters the system at time $t = r_a$ (and no other requests are pending), three cases can occur:

1. $C_a \leq c_s(t)$. Hence, J_a completes at time $f_a = t + C_a$.
2. $C_a > c_s(t)$ and the capacity is completely discharged within the current period. In this case, a portion $\Delta_a = c_s(t)$ of J_a is executed in the current server period.
3. $C_a > c_s(t)$, but the period ends before the capacity is completely discharged. In this case, the portion of J_a executed in the current server period is $\Delta_a = G_a T_s - r_a$, where $G_a = \lceil t/T_s \rceil$.

In the last two cases, depicted in Figure 5.10, the portion of J_a executed in the current server period can be computed as

$$\Delta_a = \min[c_s(t), (G_a T_s - r_a)].$$

Using the same notation introduced during polling analysis, the finishing time f_a of request J_a can then be derived as follows (see Figure 5.11):

$$f_a = \begin{cases} r_a + C_a & \text{if } C_a \leq c_s(t) \\ (F_a + G_a)T_s + R_a & \text{otherwise,} \end{cases}$$

where

$$\begin{cases} F_a = \left\lfloor \frac{C_a - \Delta_a}{C_s} \right\rfloor \\ G_a = \left\lceil \frac{r_a}{T_s} \right\rceil \\ R_a = C_a - \Delta_a - F_a C_s. \end{cases}$$

Thus, the schedulability of a single aperiodic request is guaranteed if and only if $f_a \leq r_a + D_a$.

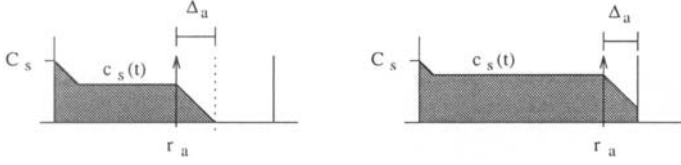


Figure 5.10 Execution of J_a in the first server period when $C_a > c_s(t)$.

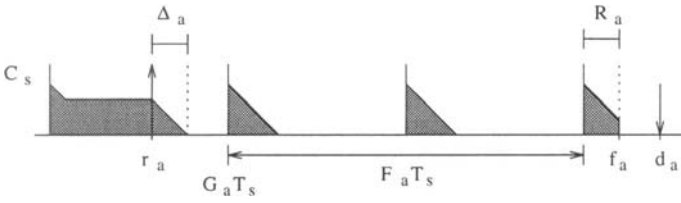


Figure 5.11 Calculation of the finishing time of J_a under DS.

To guarantee a set of firm aperiodic requests note that, at any time t , the total aperiodic computation that has to be served in any interval $[t, d_k]$ is equal to the sum of the remaining processing times $c_i(t)$ of the tasks with deadline $d_i \leq d_k$; that is,

$$C_{ape}(t, d_k) = \sum_{i=1}^k c_i(t).$$

And using the same approach adopted for the Polling Server, we define:

$$\begin{cases} G_k = \left\lceil \frac{t}{T_s} \right\rceil \\ \Delta_k = \min[c_s(t), (G_k T_s - r_a)] \\ F_k = \left\lfloor \frac{C_{ape}(t, d_k) - \Delta_k}{C_s} \right\rfloor \\ R_k = C_{ape}(t, d_k) - \Delta_k - F_k C_s. \end{cases}$$

Hence, the finishing time of the k th request is

$$f_k = \begin{cases} t + C_{ape} & \text{if } C_{ape} \leq c_s(t) \\ (F_k + G_k)T_s + R_k & \text{otherwise.} \end{cases}$$

Thus, a set of firm aperiodic requests is guaranteed at time t , if and only if

$$f_k \leq d_k \quad \forall k = 1, \dots, n.$$

5.5 PRIORITY EXCHANGE

The *Priority Exchange* (PE) algorithm is a scheduling technique introduced by Lehoczky, Sha, and Strosnider in [LSS87] for servicing a set of soft aperiodic requests along with a set of hard periodic tasks. With respect to DS, PE has a slightly worse performance in terms of aperiodic responsiveness but provides a better schedulability bound for the periodic task set.

Like DS, the PE algorithm uses a periodic server (usually at a high priority) for servicing aperiodic requests. However, it differs from DS in the manner in which the capacity is preserved. Unlike DS, PE preserves its high-priority capacity by exchanging it for the execution time of a lower-priority periodic task.

At the beginning of each server period, the capacity is replenished at its full value. If aperiodic requests are pending and the server is the ready task with the highest priority, then the requests are serviced using the available capacity; otherwise C_s is exchanged for the execution time of the active periodic task with the highest priority.

When a priority exchange occurs between a periodic task and a PE server, the periodic task executes at the priority level of the server while the server accumulates a capacity at the priority level of the periodic task. Thus, the periodic task advances its execution, and the server capacity is not lost but preserved at a lower priority. If no aperiodic requests arrive to use the capacity, priority exchange continues with other lower-priority tasks until either the capacity is used for aperiodic service or it is degraded to the priority level of background processing. Since the objective of the PE algorithm is to provide high responsiveness to aperiodic requests, all priority ties are broken in favor of aperiodic tasks.

Figure 5.12 illustrates an example of aperiodic scheduling using the PE algorithm. In this example, the PE server is created with a period $T_s = 5$ and a capacity $C_s = 1$. Since the aperiodic time managed by the PE algorithm can be exchanged with all periodic tasks, the capacity accumulated at each priority level as a function of time is represented in overlapping with the schedule of the corresponding periodic task. In particular, the first timeline of Figure 5.12 shows the aperiodic requests arriving in the system, the second timeline visualizes the capacity available at PE's priority, whereas the third and the fourth ones show the capacities accumulated at the corresponding priority levels as a consequence of the priority exchange mechanism.

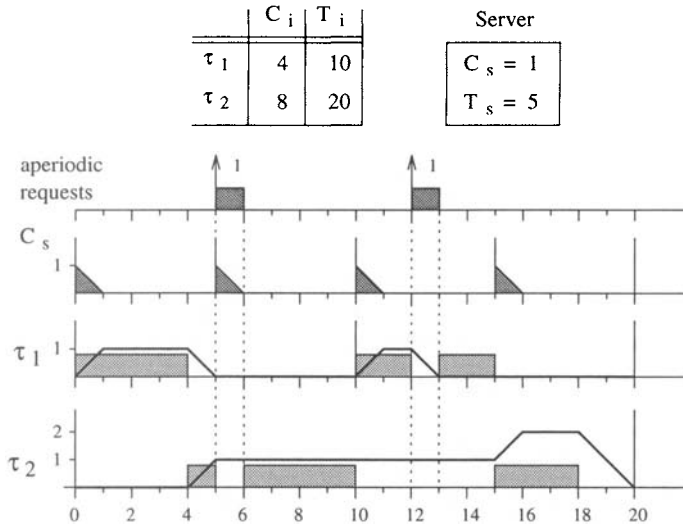


Figure 5.12 Example of aperiodic service under a PE server.

At time $t = 0$, the PE server is brought at its full capacity, but no aperiodic requests are pending, so C_s is exchanged with the execution time of task τ_1 . As a result, τ_1 advances its execution and the server accumulates one unit of time at the priority level of τ_1 . At time $t = 4$, τ_1 completes and τ_2 begins to execute. Again, since no aperiodic tasks are pending, another exchange takes place between τ_1 and τ_2 . At time $t = 5$, the capacity is replenished at the server priority, and it is used to execute the first aperiodic request. At time $t = 10$, C_s is replenished at the highest priority, but it is degraded to the priority level of τ_1 for lack of aperiodic tasks. At time $t = 12$, the capacity accumulated at the priority level of τ_1 is used to execute the second aperiodic request. At time $t = 15$, a new high-priority replenishment takes place, but the capacity is exchanged with the execution time of τ_2 . Finally, at time $t = 18$, the remaining capacity accumulated at the priority level of τ_2 is gradually discarded because no tasks are active.

Note that the capacity overlapped to the schedule of a periodic task indicates, at any instant, the amount of time by which the execution of that task is advanced with respect to the case of no exchange.

Another example of aperiodic scheduling under the PE algorithm is depicted in Figure 5.13. Here, at time $t = 5$, the capacity of the server immediately

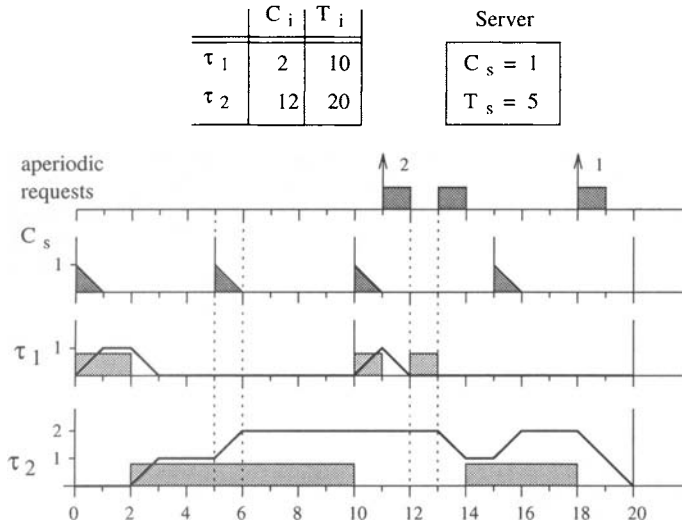


Figure 5.13 Example of aperiodic service under a PE server.

degrades down to the lowest-priority level of τ_2 , since no aperiodic requests are pending and τ_1 is idle. At time $t = 11$, when request J_1 arrives, it is interesting to observe that the first unit of computation time is immediately executed by using the capacity accumulated at the priority level of τ_1 . Then, since the remaining capacity is available at the lowest-priority level and τ_1 is still active, J_1 is preempted by τ_1 and is resumed at time $t = 13$, when τ_1 completes.

5.5.1 Schedulability analysis

The schedulability bound for a set of periodic tasks running along with a Priority Exchange server is derived with the same technique used for the Deferrable Server. The least upper bound of the processor utilization factor in the presence of PE is calculated by assuming that PE is the highest-priority task in the system. To simplify the computation of the bound, the worst-case relations among the tasks is first determined, and then the lower bound is computed against the worst-case model [LSS87].

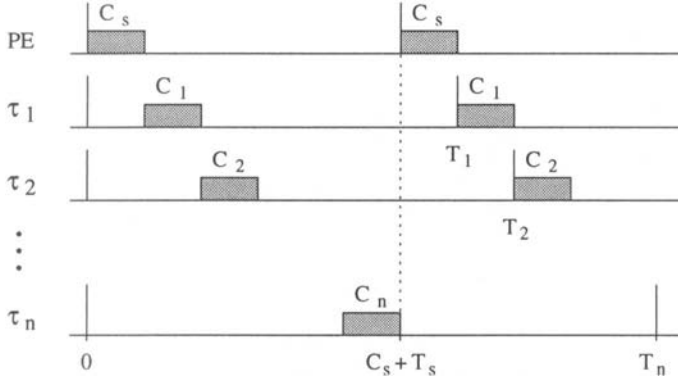


Figure 5.14 Worst-case tasks' relations under Priority Exchange.

Calculation of U_{lub} for PE+RM

Consider a set of n periodic tasks, τ_1, \dots, τ_n , ordered by increasing periods, and a PE server with a higher priority. The worst-case phasing and period relations for the periodic tasks are the same as the ones derived for the RM analysis; hence, $T_s < T_n < 2T_s$. The only difference with DS is that a PE server can execute at most two times within the period of the highest-priority periodic task. Hence, the worst-case situation for a set of periodic tasks that fully utilize the processor is the one illustrated in Figure 5.14, where tasks are characterized by the following parameters:

$$\begin{cases} C_s = T_1 - T_s \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \dots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_s - C_s - \sum_{i=1}^{n-1} C_i = 2T_s - T_n. \end{cases}$$

The resulting utilization is then

$$\begin{aligned} U &= \frac{C_s}{T_s} + \frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} = \\ &= U_s + \frac{T_2 - T_1}{T_1} + \dots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{2T_s - T_n}{T_n} = \\ &= U_s + \frac{T_2}{T_1} + \dots + \frac{T_n}{T_{n-1}} + \left(\frac{2T_s}{T_1}\right) \frac{T_1}{T_n} - n. \end{aligned}$$

Defining

$$\begin{cases} R_s = T_1/T_s \\ R_i = T_{i+1}/T_i \\ K = 2T_s/T_1 = 2/R_s \end{cases}$$

and noting that

$$R_1 R_2 \dots R_{n-1} = \frac{T_n}{T_1},$$

the utilization factor may be written as

$$U = U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{R_1 R_2 \dots R_{n-1}} - n.$$

Since this is the same expression obtained for DS, the least upper bound is

$$U_{lub} = U_s + n(K^{1/n} - 1). \quad (5.5)$$

Equation (5.5) differs from equation (5.1) only for the value of K . And noting that

$$U_s = \frac{C_s}{T_s} = \frac{T_1 - T_s}{T_s} = R_s - 1,$$

K can be rewritten as

$$K = \frac{2}{R_s} = \frac{2}{U_s + 1}.$$

Thus, finally

$$U_{lub} = U_s + n \left[\left(\frac{2}{U_s + 1} \right)^{1/n} - 1 \right]. \quad (5.6)$$

Taking the limit as $n \rightarrow \infty$, we find the worst-case bound as a function of U_s to be given by

$$\lim_{n \rightarrow \infty} U_{lub} = U_s + \ln \left(\frac{2}{U_s + 1} \right). \quad (5.7)$$

Thus, given a set of n periodic tasks and a Priority Exchange server with utilization factors U_p and U_s , respectively, the schedulability of the periodic task set is guaranteed under RM if

$$U_p + U_s \leq U_s + \ln \left(\frac{2}{U_s + 1} \right)$$

that is, if

$$U_p \leq \ln \left(\frac{2}{U_s + 1} \right). \quad (5.8)$$

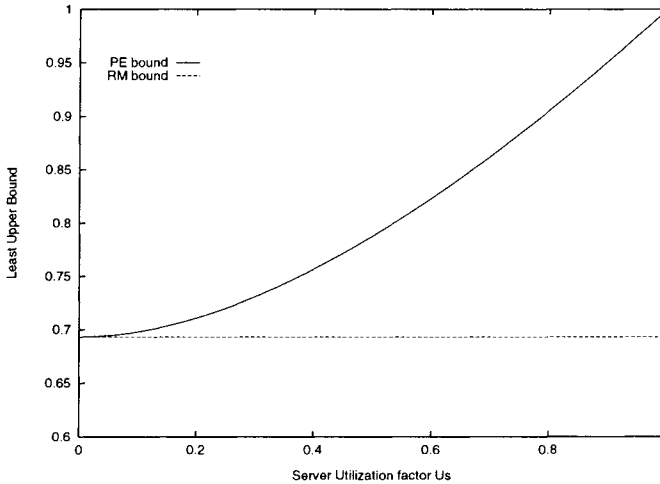


Figure 5.15 Schedulability bound for periodic tasks and PE as a function of the server utilization factor U_s .

A plot of equation (5.7) as a function of U_s is shown in Figure 5.15. For comparison, the RM bound is also reported in the plot. Notice that the schedulability test expressed in equation (5.8) is also valid for the Polling Server and, in general, for all servers that behave like a periodic task.

5.5.2 PE versus DS

The DS and the PE algorithms represent two alternative techniques for enhancing aperiodic responsiveness over traditional background and polling approaches. Here, these techniques are compared in terms of performance, schedulability bound, and implementation complexity, in order to help a system designer in selecting the most appropriate method for a particular real-time application.

The DS algorithm is much simpler to implement than the PE algorithm, because it always maintains its capacity at the original priority level and never exchanges its execution time with lower-priority tasks, as the PE algorithm does. The additional work required by PE to manage and track priority exchanges increases the overhead of PE with respect to DS, especially when the number of periodic tasks is large. On the other hand, DS does pay schedulability penalty for its simplicity in terms of a lower utilization bound. This means

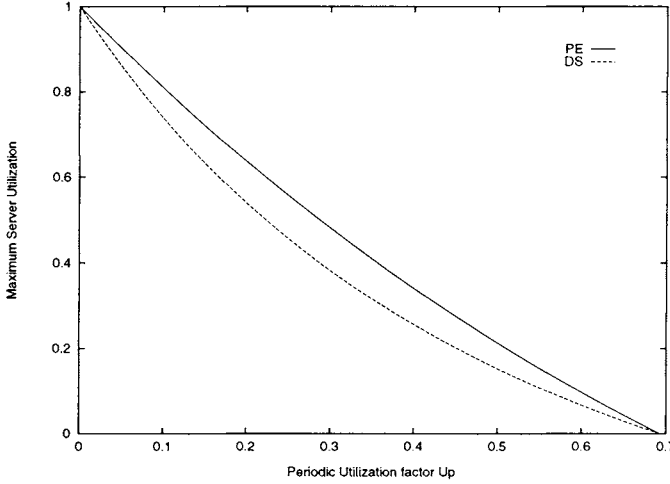


Figure 5.16 Maximum server utilization as a function of the periodic load.

that, for a given periodic load U_p , the maximum size of a DS server that can still guarantee the periodic tasks is smaller than the maximum size of a PE server.

The maximum size of a DS and a PE server as a function of U_p can easily be derived from the corresponding schedulability tests computed above. For example, from the DS schedulability test expressed in equation (5.4), the maximum utilization for DS turns out to be

$$U_{DS}^* = \frac{2 - e^{U_p}}{2e^{U_p} - 1}, \quad (5.9)$$

whereas, from equation (5.8), the maximum PE utilization is

$$U_{PE}^* = \frac{2 - e^{U_p}}{e^{U_p}}. \quad (5.10)$$

A plot of these two equations as a function of U_p is shown in Figure 5.16. Notice that, when $U_p = 0.6$, the maximum utilization for PE is 10%, whereas DS utilization cannot be greater than 7%. If instead $U_p = 0.3$, PE can have 48% utilization, while DS cannot go over 38%. The performance of the two algorithms in terms of average aperiodic response times is shown in Section 5.9.

As far as firm aperiodic tasks are concerned, the schedulability analysis under PE is much more complex than under DS. This is due to the fact that, in

general, when an aperiodic request is handled by the PE algorithm, the server capacity can be distributed among $n + 1$ priority levels. Hence, calculating the finishing time of the request might require the construction of the schedule for all the periodic tasks up to the aperiodic deadline.

5.6 SPORADIC SERVER

The *Sporadic Server* (SS) algorithm is another technique, proposed by Sprunt, Sha, and Lehoczky in [SSL89], which allows to enhance the average response time of aperiodic tasks without degrading the utilization bound of the periodic task set.

The SS algorithm creates a high-priority task for servicing aperiodic requests and, like DS, preserves the server capacity at its high-priority level until an aperiodic request occurs. However, SS differs from DS in the way it replenishes its capacity. Whereas DS and PE periodically replenish their capacity to its full value at the beginning of each server period, SS replenishes its capacity only after it has been consumed by aperiodic task execution.

In order to simplify the description of the replenishment method used by SS, the following terms are defined:

- P_{exe} It denotes the priority level of the task which is currently executing.
- P_s It denotes the priority level associated with SS.
- Active** SS is said to be *active* when $P_{exe} \geq P_s$.
- Idle** SS is said to be *idle* when $P_{exe} < P_s$.
- RT** It denotes the *replenishment time* at which the SS capacity will be replenished.
- RA** It denotes the *replenishment amount* that will be added to the capacity at time RT.

Using this terminology, the capacity C_s consumed by aperiodic requests is replenished according to the following rule:

- The replenishment time RT is set as soon as SS becomes active and $C_s > 0$. Let t_A be such a time. The value of RT is set equal to t_A plus the server period ($RT = t_A + T_s$).
- The replenishment amount RA to be done at time RT is computed when SS becomes idle or C_s has been exhausted. Let t_I be such a time. The value of RA is set equal to the capacity consumed within the interval $[t_A, t_I]$.

An example of medium-priority SS is shown in Figure 5.17. To facilitate the understanding of the replenishment rule, the intervals in which SS is active are also shown. At time $t = 0$, the highest-priority task τ_1 is scheduled, and SS becomes active. Since $C_s > 0$, a replenishment is set at time $RT_1 = t + T_s = 10$. At time $t = 1$, τ_1 completes, and, since no aperiodic requests are pending, SS becomes idle. Note that no replenishment takes place at time $RT_1 = 10$ ($RA_1 = 0$) because no capacity has been consumed in the interval $[0, 1]$. At time $t = 4$, the first aperiodic request J_1 arrives, and, since $C_s > 0$, SS becomes active and the request receives immediate service. As a consequence, a replenishment is set at $RT_2 = t + T_s = 14$. Then, J_1 is preempted by τ_1 at $t = 5$, is resumed at $t = 6$ and is completed at $t = 7$. At this time, the replenishment amount to be done at RT_2 is set equal to the capacity consumed in $[4, 7]$; that is, $RA_2 = 2$.

Notice that during preemption intervals SS stays active. This allows to perform a single replenishment, even if SS provides a discontinuous service for aperiodic requests.

At time $t = 8$, SS becomes active again and a new replenishment is set at $RT_3 = t + T_s = 18$. At $t = 11$, SS becomes idle and the replenishment amount to be done at RT_3 is set to $RA_3 = 2$.

Figure 5.18 illustrates another example of aperiodic service in which SS is the highest-priority task. Here, the first aperiodic request arrives at time $t = 2$ and consumes the whole server capacity. Hence, a replenishment amount $RA_1 = 2$ is set at $RT_1 = 10$. The second request arrives when $C_s = 0$. In this case, the replenishment time RT_2 is set as soon as the capacity becomes greater than zero. Since this occurs at time $t = 10$, the next replenishment is set at time $RT_2 = 18$. The corresponding replenishment amount is established when J_2 completes and is equal to $RA_2 = 2$.

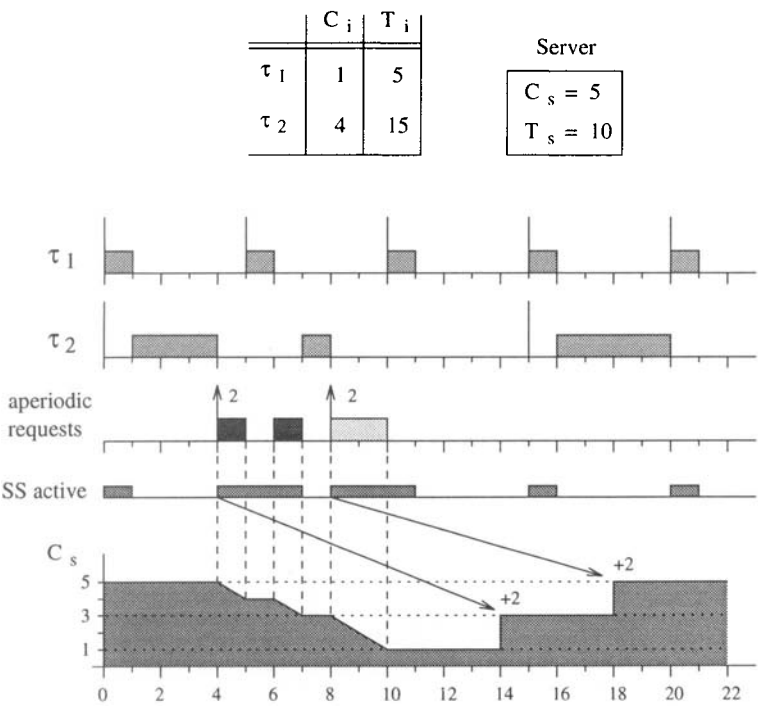


Figure 5.17 Example of a medium-priority Sporadic Server.

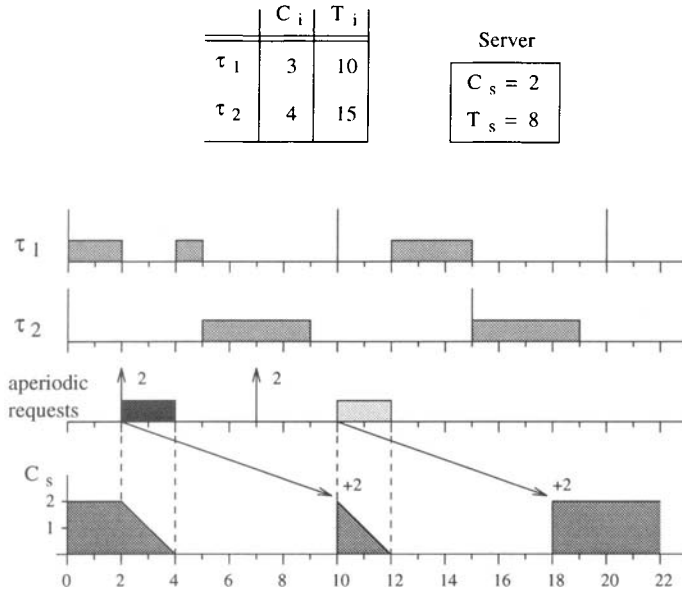


Figure 5.18 Example of a high-priority Sporadic Server.

5.6.1 Schedulability analysis

The Sporadic Server violates one of the basic assumptions governing the execution of a standard periodic task. This assumption requires that once a periodic task is the highest-priority task that is ready to execute, it must execute. Like DS, in fact, SS defers its execution and preserves its capacity when no aperiodic requests are pending. However, we show that the replenishment rule used in SS compensates for any deferred execution and, from a scheduling point of view, SS can be treated as a normal periodic task with a period T_s and an execution time C_s . In particular, the following theorem holds [SSL89]:

Theorem 5.1 (Sprunt-Sha-Lehoczky) *A periodic task set that is schedulable with a task τ_i is also schedulable if τ_i is replaced by a Sporadic Server with the same period and execution time.*

Proof. The theorem is proved by showing that for any type of service, SS exhibits an execution behavior equivalent to one or more periodic tasks. Let t_A be the time at which C_s is full and SS becomes active, and let t_I be the time at which SS becomes idle, such that $[t_A, t_I]$ is a continuous interval during which

SS remains active. The execution behavior of the server in the interval $[t_A, t_I]$ can be described by one of the following three cases (see Figure 5.19):

1. No capacity is consumed.
2. The server capacity is totally consumed.
3. The server capacity is partially consumed.

- Case 1.** If no requests arrive in $[t_A, t_I]$, SS preserves its capacity and no replenishments can be performed before time $t_I + T_s$. This means that at most C_s units of aperiodic time can be executed in $[t_A, t_I + T_s]$. Hence, the SS behavior is identical to a periodic task $\tau_s(C_s, T_s)$ whose release time is delayed from t_A to t_I . As proved in Chapter 4 for RM, delaying the release of a periodic task cannot increase the response time of the other periodic tasks; therefore, this case does not jeopardize schedulability.
- Case 2.** If C_s is totally consumed in $[t_A, t_I]$, a replenishment of C_s units of time will occur at time $t_A + T_s$. Hence, SS behaves like a periodic task with period T_s and execution time C_s released at time t_A .
- Case 3.** If C_s is partially consumed in $[t_A, t_I]$, a replenishment will occur at time $t_A + T_s$, and the remaining capacity is preserved for future requests. Let C_R be the capacity consumed in $[t_A, t_I]$. In this case, the behavior of the server is equivalent to two periodic tasks, τ_x and τ_y , with periods $T_x = T_y = T_s$, and execution times $C_x = C_R$ and $C_y = C_s - C_R$, such that τ_x is released at t_A and τ_y is delayed until t_I . As in Case 1, the delay of τ_y has no schedulability effects.

Since in any servicing situation SS can be represented by one or more periodic tasks with period T_s and total execution time equal to C_s , the contribution of SS in terms of processor utilization is equal to $U_s = C_s/T_s$. Hence, from a schedulability point of view, SS can be replaced by a periodic task having the same utilization factor. \square

Since SS behaves like a normal periodic task, the periodic task set can be guaranteed by the same schedulability test derived for PE. Hence, a set Γ of n

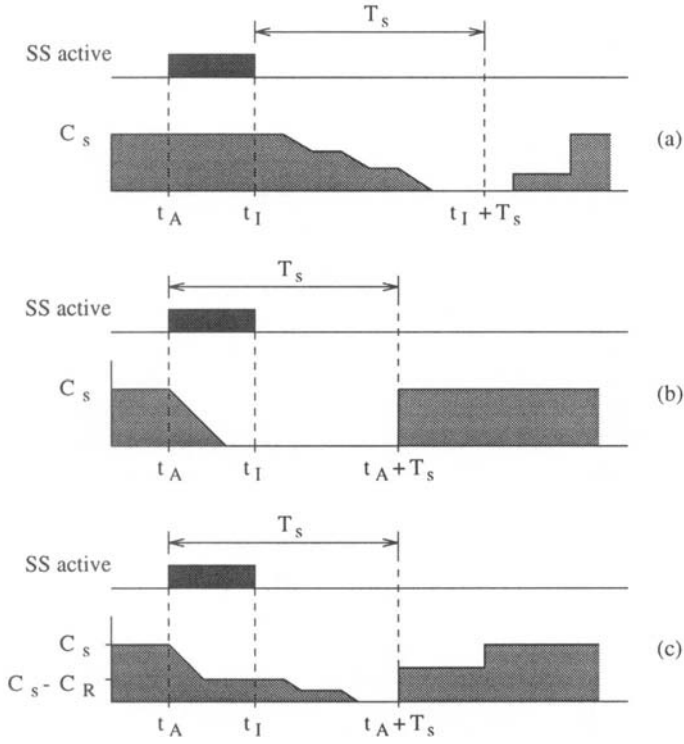


Figure 5.19 Possible SS behavior during active intervals: **a.** C_s is not consumed; **b.** C_s is totally consumed; **c.** C_s is partially consumed.

periodic tasks with utilization factor U_p scheduled along with a Sporadic Server with utilization U_s are schedulable under RM if

$$U_p \leq n \left[\left(\frac{2}{U_s + 1} \right)^{1/n} - 1 \right]. \quad (5.11)$$

For large n , Γ is schedulable if

$$U_p \leq \ln \left(\frac{2}{U_s + 1} \right) \quad (5.12)$$

For a given U_p , the maximum server size that guarantees the schedulability of the periodic tasks is

$$U_{SS}^* = 2 \left(\frac{U_p}{n} + 1 \right)^{-n} - 1, \quad (5.13)$$

and for large n it becomes

$$U_{SS}^* = \frac{2}{e^{U_p}} - 1. \quad (5.14)$$

As far as firm aperiodic tasks are concerned, the schedulability analysis under SS is not simple because, in general, the server capacity can be fragmented in a lot of small pieces of different size, available at different times according to the replenishment rule. As a consequence, calculating the finishing time of an aperiodic request requires to keep track of all the replenishments that will occur until the task deadline.

5.7 SLACK STEALING

The *Slack Stealing* algorithm is another aperiodic service technique, proposed by Lehoczky and Ramos-Thuel in [LRT92], which offers substantial improvements in response time over the previous service methods (PE, DS, and SS). Unlike these methods, the Slack Stealing algorithm does not create a periodic server for aperiodic task service. Rather it creates a passive task, referred to as the *Slack Stealer*, which attempts to make time for servicing aperiodic tasks by “stealing” all the processing time it can from the periodic tasks without causing their deadlines to be missed. This is equivalent to stealing slack from the periodic tasks. We recall that, if $c_i(t)$ is the remaining computation time at time t , the slack of a task τ_i is

$$slack_i(t) = d_i - t - c_i(t).$$

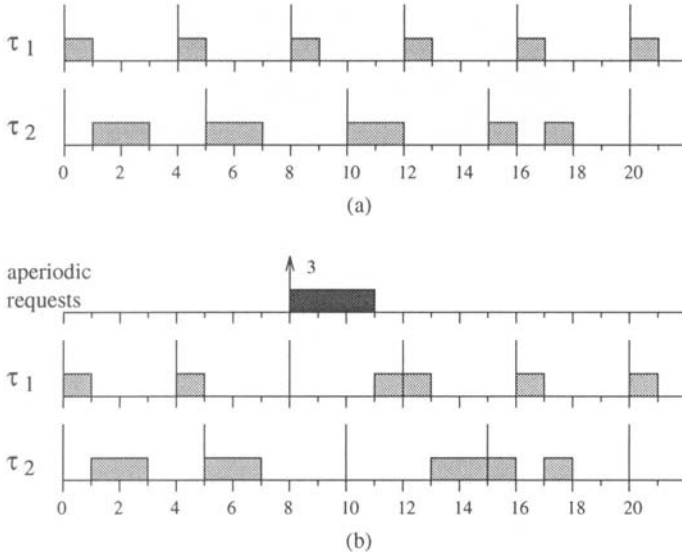


Figure 5.20 Example of Slack Stealer behavior: **a.** when no aperiodic requests are pending; **b.** when an aperiodic request of three units arrives at time $t = 8$.

The main idea behind slack stealing is that, typically, there is no benefit in early completion of the periodic tasks. Hence, when an aperiodic request arrives, the Slack Stealer steals all the available slack from periodic tasks and uses it to execute aperiodic requests as soon as possible. If no aperiodic requests are pending, periodic tasks are normally scheduled by RM. Similar algorithms based on slack stealing have been proposed by other authors [RTL93, DTB93, TLS95].

Figure 5.20 shows the behavior of the Slack Stealer on a set of two periodic tasks, τ_1 and τ_2 , with periods $T_1 = 4$, $T_2 = 5$ and execution times $C_1 = 1$, $C_2 = 2$. In particular, Figure 5.20a shows the schedule produced by RM when no aperiodic tasks are processed, whereas Figure 5.20b illustrates the case in which an aperiodic request of three units arrives at time $t = 8$ and receives immediate service. In this case, a slack of three units is obtained by delaying the third instance of τ_1 and τ_2 .

Notice that, in the example of Figure 5.20, no other server algorithms (DS, PE, or SS) can schedule the aperiodic requests at the highest priority and still guarantee the periodic tasks. For example, since $U_p = 1/4 + 2/5 = 0.65$, the

maximum utilization factor for a sporadic server to guarantee the schedulability of the periodic task set is (see equation (5.13)):

$$U_{SS}^* = 2 \left(\frac{U_p}{2} + 1 \right)^{-2} - 1 \simeq 0.14.$$

This means that, even with $C_s = 1$, the shortest server period that can be set with this utilization factor is $T_s = \lceil C_s / U_s \rceil = 8$, which is greater than both task periods. Thus, the execution of the server would be equivalent to a background service, and the aperiodic request would be completed at time 15.

5.7.1 Schedulability analysis

In order to schedule an aperiodic request $J_a(r_a, C_a)$ according to the Slack-Stealing algorithm, we need to determine the earliest time t such that at least C_a units of slack are available in $[r_a, t]$. The computation of the slack is carried out through the use of a *slack function* $A(s, t)$, which returns the maximum amount of computation time that can be assigned to aperiodic requests in the interval $[s, t]$ without compromising the schedulability of periodic tasks.

Figure 5.21 shows the slack function at time $s = 0$ for the periodic task set considered in the previous example. For a given s , $A(s, t)$ is a non-decreasing step function defined over the hyperperiod, with jump points corresponding to the beginning of the intervals where the slack is available. As s varies, the slack function needs to be recomputed, and this requires a relatively large amount of calculation, especially for long hyperperiods. Figure 5.22 shows how the slack function $A(s, t)$ changes at time $s = 6$ for the same periodic task set.

According to the original algorithm proposed by Lehoczky and Ramos-Thuel [LRT92], the slack function at time $s = 0$ is precomputed and stored in a table. During runtime, the actual function $A(s, t)$ is then computed by updating $A(0, t)$ based on the periodic execution time, the aperiodic service time, and the idle time. The complexity for computing the current slack from the table is $O(n)$, where n is the number of periodic tasks; however, depending on the periods of the tasks, the size of the table can be too large for practical implementations.

A *dynamic* method of computing slack has been proposed by Davis, Tindell, and Burns in [DTB93]. According to this algorithm, the available slack is computed whenever an aperiodic request enters the system. This method is more complex than the previous *static* approach, but it requires much less memory

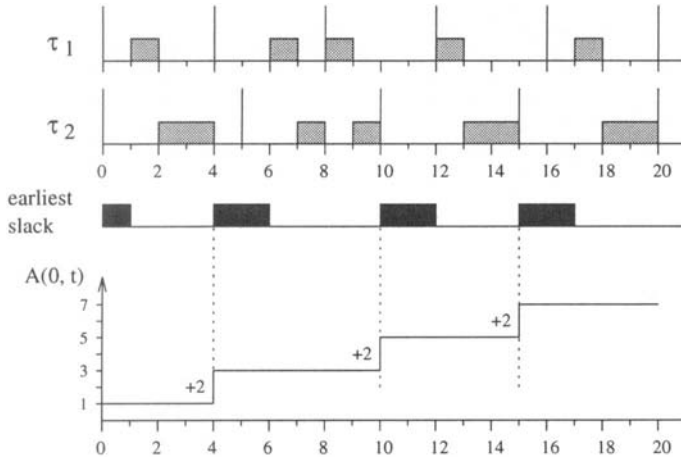


Figure 5.21 Slack function at time $s = 0$ for the periodic task set considered in the previous example.

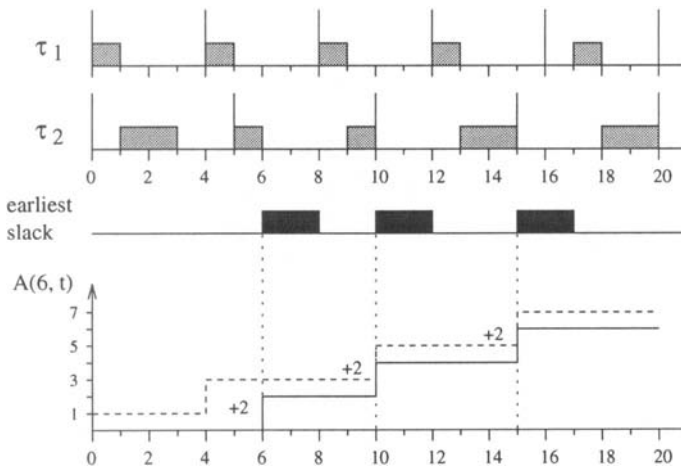


Figure 5.22 Slack function at time $s = 6$ for the periodic task set considered in the previous example.

and allows handling of periodic tasks with release jitter or synchronization requirements. Finally, a more efficient algorithm for computing the slack function has been proposed by Tia, Liu, and Shankar in [TLS95].

The Slack-Stealing algorithm has also been extended by Ramos-Thuel and Lehoczy [RTL93] to guarantee firm aperiodic tasks.

5.8 NON-EXISTENCE OF OPTIMAL SERVERS

The Slack Stealer always advances all available slack as much as possible and uses it to execute the pending aperiodic tasks. For this reason, it originally was considered an optimal algorithm; that is, capable of minimizing the response time of every aperiodic request. Unfortunately, the Slack Stealer is not optimal because to minimize the response time of an aperiodic request, it is sometimes necessary to schedule it at a later time even if slack is available at the current time. Indeed, Tia, Liu, and Shankar [TLS95] proved that, if periodic tasks are scheduled using a fixed-priority assignment, no algorithm can minimize the response time of every aperiodic request and still guarantee the schedulability of the periodic tasks.

Theorem 5.2 (Tia-Liu-Shankar) *For any set of periodic tasks ordered on a given fixed-priority scheme and aperiodic requests ordered according to a given aperiodic queueing discipline, there does not exist any valid algorithm that minimizes the response time of every soft aperiodic request.*

Proof. Consider a set of three periodic tasks with $C_1 = C_2 = C_3 = 1$ and $T_1 = 3$, $T_2 = 4$ and $T_3 = 6$, whose priorities are assigned based on the RM algorithm. Figure 5.23a shows the schedule of these tasks when no aperiodic requests are processed.

Now consider the case in which an aperiodic request J_1 , with $C_{a_1} = 1$, arrives at time $t = 2$. At this point, any algorithm has two choices:

1. Do not schedule J_1 at time $t = 2$. In this case, the response time of J_1 will be greater than 1 and, thus, it will not be the minimum.

2. Schedule J_1 at time $t = 2$. In this case, assume that another request J_2 , with $C_{a_2} = 1$, arrives at time $t = 3$. Since no slack time is available in the interval $[3, 6]$, J_2 can start only at $t = 6$ and finish at $t = 7$. This situation is shown in Figure 5.23b.

However, the response time of J_2 achieved in case 2 is not the minimum possible. In fact, if J_1 were scheduled at time $t = 3$, another unit of slack would have been available at time $t = 4$, thus J_2 would have been completed at time $t = 5$. This situation is illustrated in Figure 5.23c.

The above example shows that it is not possible for any algorithm to minimize the response times of J_1 and J_2 simultaneously. If J_1 is scheduled immediately, then J_2 will not be minimized. On the other hand, if J_1 is delayed to minimize J_2 , then J_1 will suffer. Hence, there is no optimal algorithm that can minimize the response time of any aperiodic request. \square

Notice that Theorem 5.2 applies both to clairvoyant and on-line algorithms since the example is applicable regardless of whether the algorithm has a priori knowledge of the aperiodic requests. The same example can be used to prove another important result on the minimization of the average response time.

Theorem 5.3 (Tia-Liu-Shankar) *For any set of periodic tasks ordered on a given fixed-priority scheme and aperiodic requests ordered according to a given aperiodic queueing discipline, there does not exist any on-line valid algorithm that minimizes the average response time of the soft aperiodic requests.*

Proof. From the example illustrated in Figure 5.23 it is easy to see that, if there is only request J_1 in each hyperperiod, then scheduling J_1 as soon as possible will yield the minimum average response time. On the other hand, if J_1 and J_2 are present in each hyperperiod, then scheduling each aperiodic request as soon as possible will not yield the minimum average response time. This means that, without a priori knowledge of the aperiodic requests' arrival, an on-line algorithm will not know when to schedule the requests. \square

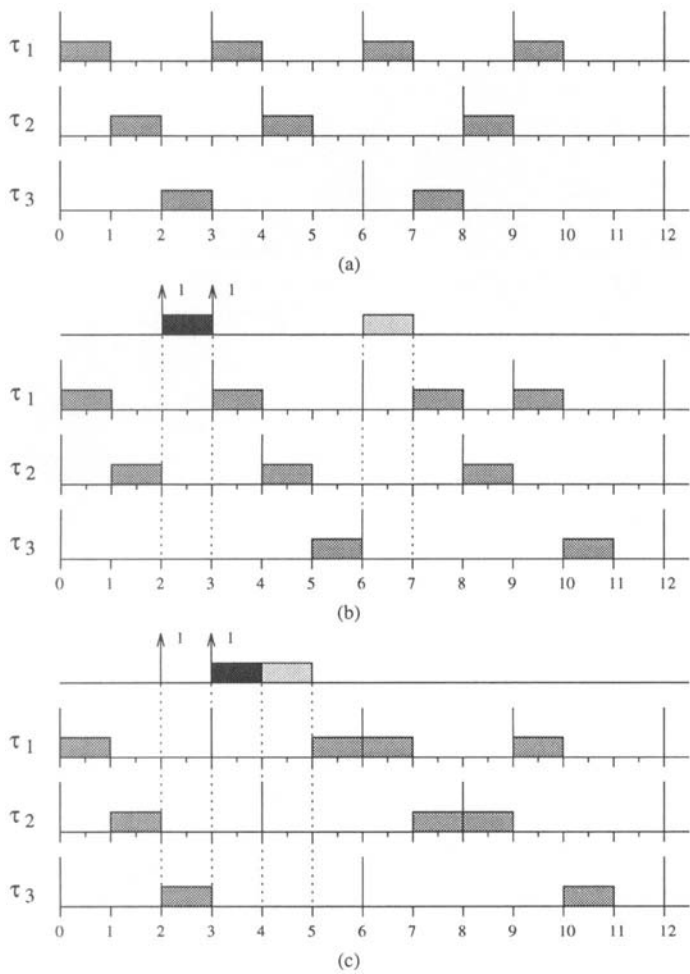


Figure 5.23 No algorithm can minimize the response time of every aperiodic request. If J_1 is minimized, J_2 is not (b). On the other hand, if J_2 is minimized, J_1 is not (c).

5.9 PERFORMANCE EVALUATION

The performance of the various algorithms described in this chapter has been compared in terms of average response times on soft aperiodic tasks. Simulation experiments have been conducted using a set of ten periodic tasks with periods ranging from 54 to 1200 units of time and utilization factor $U_p = 0.69$. The aperiodic load was varied across the unused processor bandwidth. The interarrival times for the aperiodic tasks were modeled using a Poisson arrival pattern with average interarrival time of 18 units of time, whereas the computation times of aperiodic requests were modeled using an exponential distribution. Periods for PS, DS, PE, and SS were set to handle aperiodic requests at the highest priority in the system (priority ties were broken in favor of aperiodic tasks). Finally, the server capacities were set to the maximum value for which the periodic tasks were schedulable.

In the plots shown in Figure 5.24, the average aperiodic response time of each algorithm is presented relative to the response time of background aperiodic service. This means that a value of 1.0 in the graph is equivalent to the average response time of background service, while an improvement over background service corresponds to a value less than 1.0. The lower the response time curve lies on the graph, the better the algorithm is for improving aperiodic responsiveness.

As can be seen from the graphs, DS, PE, and SS provide a substantial reduction in the average aperiodic response time compared to background and polling service. In particular, a better performance is achieved with short and frequent requests. This can be explained by considering that, in most of the cases, short tasks do not use the whole server capacity and can finish within the current server period. On the other hand, long tasks protract their completion because they consume the whole server capacity and have to wait for replenishments.

Notice that average response times achieved by SS are slightly higher than those obtained by DS and PE. This is mainly due to the different replenishment rule used by the algorithms. In DS and PE, the capacity is always replenished at its full value at the beginning of every server period, while in SS it is replenished T_s units of time after consumption. Thus, in the average, when the capacity is exhausted, waiting for replenishment in SS is longer than waiting in DS or in PE.

Figure 5.25 shows the performance of the Slack-Stealing algorithm with respect to background service, Polling, and SS. The performance of DS and PE is not

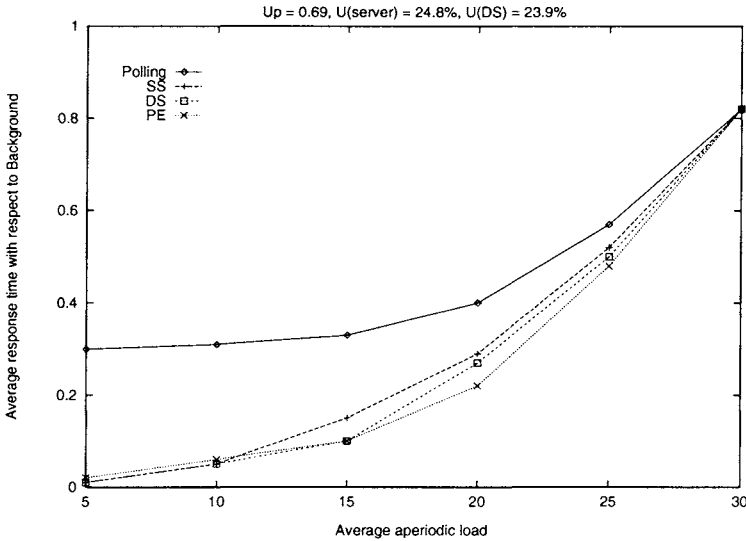


Figure 5.24 Performance results of PS, DS, PE, and SS.

shown because it is very similar to the one of SS. Unlike the previous figure, in this graph the average response times are not reported relative to background, but are directly expressed in time units. As we can see, the Slack-Stealing algorithm outperforms all the other scheduling algorithms over the entire range of aperiodic load. However, the largest performance gain of the Slack Stealer over the other algorithms occurs at high aperiodic loads, when the system reaches the upper limit as imposed by the total resource utilization.

Other simulation results can be found in [LSS87] for Polling, PE, and DS, in [SSL89] for SS, and in [LRT92] for the Slack-Stealing algorithm.

5.10 SUMMARY

The algorithms presented in this chapter can be compared not only in terms of performance but also in terms of computational complexity, memory requirement, and implementation complexity. In order to select the most appropriate service method for handling soft aperiodic requests in a hard real-time environment, all these factors should be considered. Figure 5.26 provides a qualitative evaluation of the algorithms presented in this chapter.

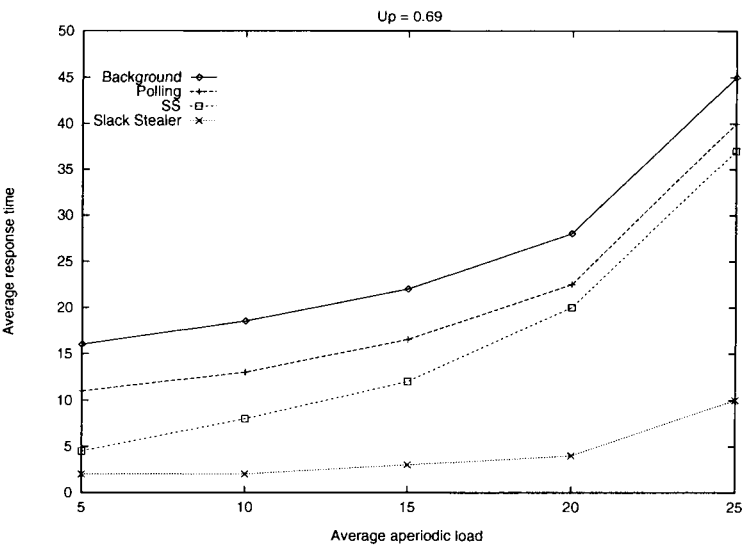





Figure 5.25 Performance of the Slack Stealer with respect to background, PS, and SS.


excellent


good


poor

























	performance	computational complexity	memory requirement	implementation complexity
Background Service				
Polling Server				
Deferrable Server				
Priority Exchange				
Sporadic Server				
Slack Stealer				

Figure 5.26 Evaluation summary of fixed-priority servers.

Exercises

- 5.1 Compute the maximum processor utilization that can be assigned to a Sporadic Server to guarantee the following periodic tasks under RM:

	τ_1	τ_2
C_i	1	2
T_i	5	8

- 5.2 Compute the maximum processor utilization that can be assigned to a Deferrable Server to guarantee the task set illustrated in Exercise 5.1.
- 5.3 Together with the periodic tasks illustrated in Exercise 5.1, schedule the following aperiodic tasks with a Polling Server having maximum utilization and intermediate priority.

	J_1	J_2	J_3
a_i	2	7	9
C_i	3	2	1

- 5.4 Solve the same scheduling problem described in Exercise 5.3, with a Sporadic Server having maximum utilization and intermediate priority.
- 5.5 Solve the same scheduling problem described in Exercise 5.3, with a Deferrable Server having maximum utilization and highest priority.
- 5.6 Solve the same scheduling problem described in Exercise 5.3, with a Priority Exchange Server having maximum utilization and highest priority.
- 5.7 Using a Sporadic Server with capacity $C_s = 2$ and period $T_s = 5$, schedule the following tasks:

periodic tasks

	τ_1	τ_2
C_i	1	2
T_i	4	6

aperiodic tasks

	J_1	J_2	J_3
a_i	2	5	10
C_i	2	1	2

- 5.8 Given the same tasks described in Exercise 5.7, compute the maximum capacity that can be assigned to a Sporadic Server with a period $T_s = 4$. Then, schedule the tasks using such a capacity.