UNIT – 4
Scheduling Aperiodic and Sporadic Jobs in Priority-Driven Systems

7.1 ASSUMPTIONS AND APPROACHES

Sporadic jobs may arrive at any instant. Their execution times may vary widely, and their deadlines are arbitrary. It is impossible for some sporadic jobs to meet their deadlines no matter what algorithm we use to schedule them. The only alternatives are (1) to reject the sporadic jobs that cannot complete in time or (2) to accept all sporadic jobs and allow some of them to complete late.

7.1.1 Objectives, Correctness, and Optimality

We assume that we are given the parameters $\{p_i\}$ and $\{e_i\}$ of all the periodic tasks and a priority-driven algorithm used to schedule them. We also assume that the operating system maintains the priority queues shown in Figure 7–1. The ready periodic jobs are placed in the periodic task queue, ordered by their priorities. Each accepted sporadic job is assigned a priority and is placed in a priority queue. Each newly arrived aperiodic job is placed in the aperiodic job queue. Newly arrived sporadic jobs are inserted into a waiting queue.

The aperiodic job and sporadic job scheduling algorithms are solutions to the following problems:

1. Based on the execution time and deadline of each newly arrived sporadic job, the scheduler decides whether to accept or reject the job. If it accepts the job, it schedules the job so that the job completes in time without causing periodic tasks and previously accepted sporadic jobs to miss their deadlines. The problems are how to do the acceptance test and how to schedule the accepted sporadic jobs.
2. The scheduler tries to complete each aperiodic job as soon as possible. The problem is how to do so without causing periodic tasks and accepted sporadic jobs to miss their deadlines.

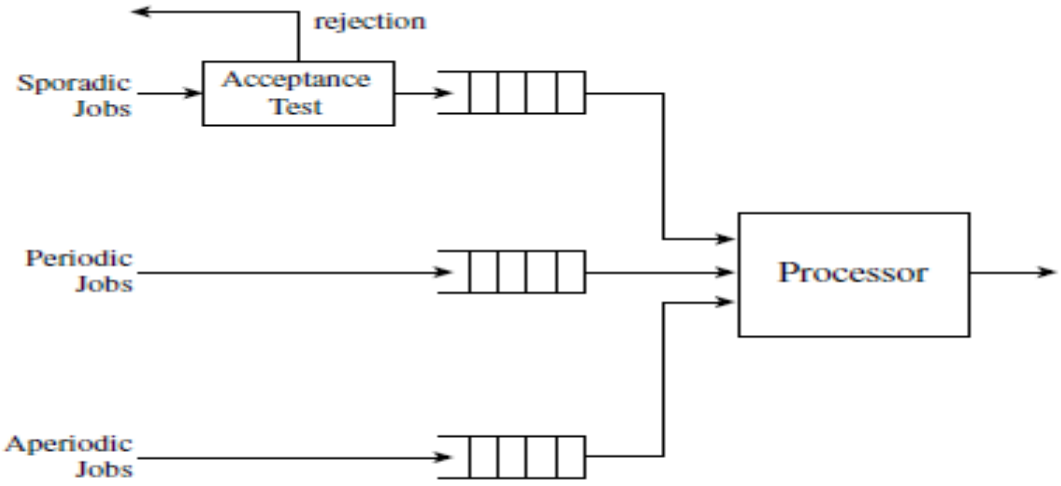


FIGURE 7–1 Priority queues maintained by the operating system.

An algorithm is ***correct*** if it produces only correct schedules of the system. By a ***correct schedule***, we mean one according to which periodic and accepted sporadic jobs never miss their deadlines. We consider only correct algorithms.

Finally, we assume that the queueing discipline used to order aperiodic jobs among themselves is given. An aperiodic job scheduling algorithm is optimal if it minimizes either the response time of the aperiodic job at the head of the aperiodic job queue.

7.1.2 Alternative Approaches

Background and Interrupt-Driven Execution versus Slack Stealing.

According to the ***background*** approach, aperiodic jobs are scheduled and executed only at times when there is no periodic or sporadic job ready for execution. Clearly this method always produces correct schedules and is simple to implement. However, the execution of aperiodic jobs may be delayed and their response times prolonged unnecessarily.

As an example, we consider the system of two periodic tasks $T_1 = (3, 1)$ and $T_2 = (10, 4)$ shown in Figure 7–2. The tasks are scheduled rate monotonically. Suppose that an aperiodic job A with execution time equal to 0.8 is released (i.e., arrives) at time 0.1. If this job is executed in the background, its execution begins after $J_{1,3}$ completes (i.e., at time 7) as shown in Figure 7–2(a). Consequently, its response time is 7.7. An obvious way to make the response times of aperiodic jobs as short as possible is to make their execution interrupt-driven.

Whenever an aperiodic job arrives, the execution of periodic tasks are interrupted, and the aperiodic job is executed. In this example, A would execute starting from 0.1 and have the shortest possible response time. The problem with this scheme is equally obvious: If aperiodic jobs always execute as soon as possible, periodic tasks may miss some deadlines. In our example, if the execution time of A were equal to 2.1, both $J_{1,1}$ and $J_{2,1}$ would miss their deadlines.

The obvious solution is to postpone the execution of periodic tasks only when it is safe to do so. From Figure 7–2(a), we see that the execution of $J_{1,1}$ and $J_{2,1}$ can be postponed by 2 units of time because they both have this much slack. By postponing the execution of $J_{1,1}$ and $J_{2,1}$, the scheduler allows aperiodic job A to execute immediately after its arrival.

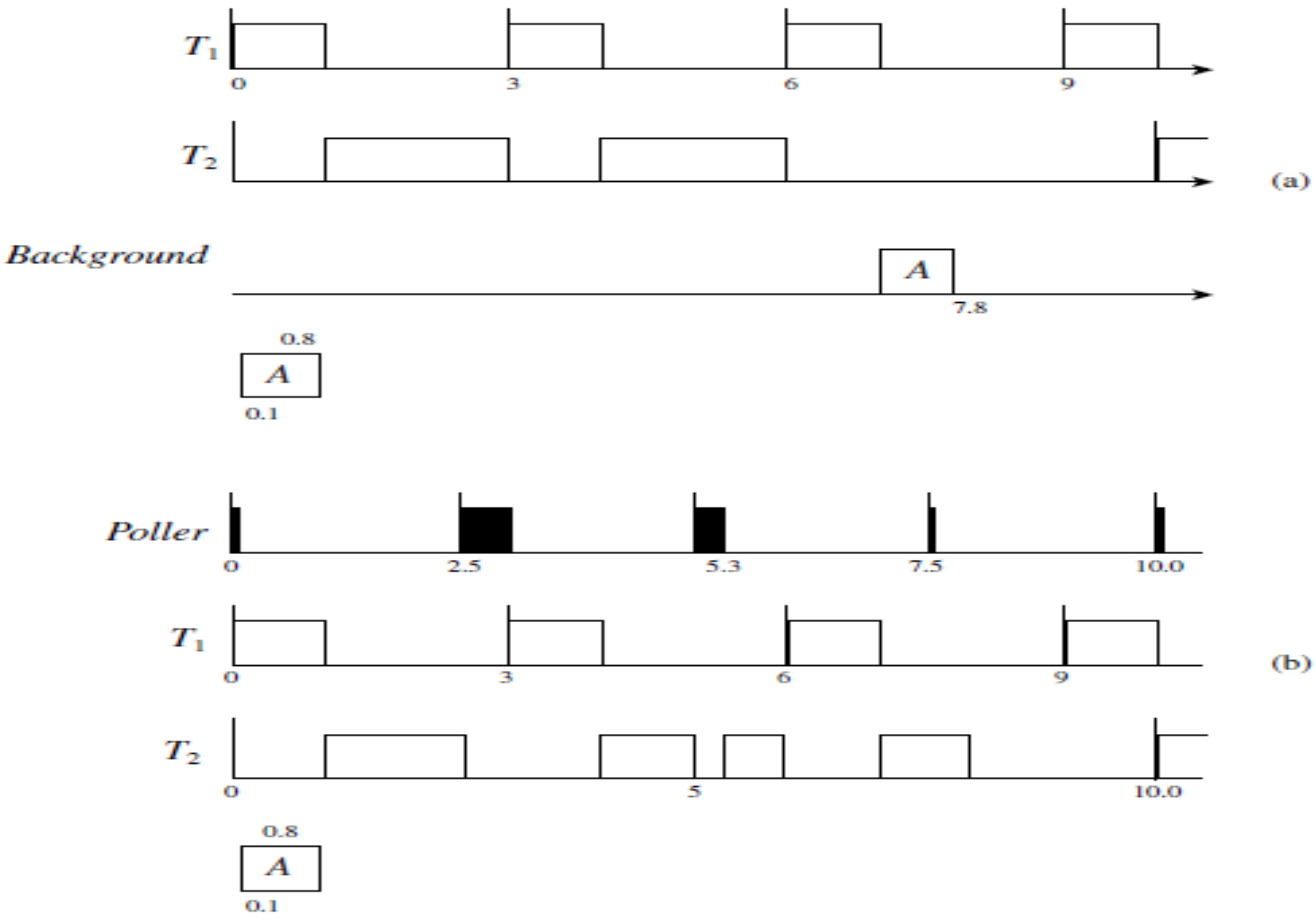


FIGURE 7–2 Commonly used methods for scheduling aperiodic jobs: $T_1 = (3, 1)$, $T_2 = (10, 4)$, poller $= (2.5, 0.5)$. (a) Background execution. (b) Polling.

The response time of *A* is equal to 0.8, which is as short as possible. The periodic job $J_{1,1}$ completes at time 1.8, and $J_{2,1}$ completes at 7.8. Both of them meet their deadlines.

Polled Executions versus Bandwidth-Preserving Servers. Polling is another commonly used way to execute aperiodic jobs. In our terminology, a ***poller* or *polling server*** *(p_s , e_s)* is a periodic task: p_s is its polling period, and e_s is its execution time. The poller is ready for execution periodically at integer multiples of p_s and is scheduled together with the periodic tasks in the system. When it executes, it examines the aperiodic job queue. If the queue is nonempty, the poller executes the job at the head of the queue. The poller suspends its execution or is suspended by the scheduler, whichever occurs sooner. It is ready for execution again at the beginning of the next polling period.

If at the beginning of a polling period the poller finds the aperiodic job queue empty, it suspends immediately. It will not be ready for execution and able to examine the queue again until the next polling period.

We call a task that behaves more or less like a periodic task and is created for the purpose of executing aperiodic jobs a ***periodic server***. A periodic server *(p_s , e_s)* is defined partially by its period *ps* and execution time e_s . The parameter e_s is called the ***execution budget*** (or simply *budget*) of the server.

The ratio $u_s = e_s/p_s$ is the ***size*** of the server. A poller *(p_s , e_s)* is a kind of periodic server. At the beginning of each period, the budget of the poller is set to e_s . We say that its budget is *replenished* (by e_s units) and call a time instant when the server budget is replenished a ***replenishment time***. We say that the periodic server is ***backlogged*** whenever the aperiodic job queue is nonempty and, hence, there is at least an aperiodic job to be executed by the server. The server

is ***idle*** when the queue is empty. The server is ***eligible for execution only when it is backlogged and has budget***. When the server is eligible, the scheduler schedules it. When the server is scheduled and executes aperiodic jobs, it ***consumes*** its budget at the rate of one per unit time. We say that the server budget becomes ***exhausted*** when the budget becomes zero.

Different kinds of periodic servers differ in how the server budget changes when the server still has budget but the server is idle. As an example, the budget of a poller becomes exhausted instantaneously whenever the poller finds the aperiodic job queue empty.

Figure 7–2(b) shows a poller in the midst of the two fixed-priority periodic tasks $T_1 = (3, 1)$ and $T_2 = (10, 4)$. The poller has period 2.5 and execution budget 0.5. It is treated by the scheduler as the periodic task $(2.5, 0.5)$ and is given the highest priority among periodic tasks. At the beginning of the first polling period, the poller's budget is replenished, but when it executes, it finds the aperiodic job queue empty. Its execution budget is consumed instantaneously, and its execution suspended immediately. The aperiodic job *A* arrives a short time later and must wait in the queue until the beginning of the second polling period when the poller's budget is replenished. The poller finds *A* at head of the queue at time 2.5 and executes the job until its execution budget is exhausted at time 3.0. Job *A* remains in the aperiodic job queue and is again executed when the execution budget of the poller is replenished at 5.0. The job completes at time 5.3, with a response time of 5.2. Since the aperiodic job queue is empty at 5.3, the budget of the poller is exhausted and the poller suspends.

This example illustrates the ***shortcoming of the polling approach***. An aperiodic job that arrives after the aperiodic job queue is examined and found empty must wait for the poller to return to examine the queue again a polling period later. If we can ***preserve*** the execution budget of the poller when it finds an empty queue and allow it to execute later in the period if any aperiodic job arrives, we may be able to shorten the response times of some aperiodic jobs.

Algorithms that improve the polling approach in this manner are called ***bandwidth preserving server*** algorithms. Bandwidth-preserving servers are periodic servers. Each type of server is defined by a set of ***consumption*** and ***replenishment*** rules. The former give the conditions under which its execution budget is preserved and consumed. The latter specify when and by how much the budget is replenished. We assume that they work as follows:.

- A backlogged bandwidth-preserving server is ready for execution when it has budget. The scheduler keeps track of the consumption of the server budget and suspends the server when the server budget is exhausted or the server becomes idle. The scheduler moves the server back to the ready queue once it replenishes the server budget.
- The server suspends itself whenever it finds the aperiodic job queue empty, that is, when it becomes idle. When the server becomes backlogged again upon of arrival of an aperiodic job, the scheduler puts the server back to the ready queue if the server has budget at the time.

7.2 DEFERRABLE SERVERS

A ***deferrable server*** is the simplest of bandwidth-preserving servers. Like a poller, the execution budget of a deferrable server with period p_s and execution budget e_s is replenished periodically with period p_s . ***Unlike a poller***, however, when a deferrable server finds no aperiodic job ready for execution, ***it preserves its budget.***

7.2.1 Operations of Deferrable Servers

Specifically, the consumption and replenishment rules that define a **deferrable server** (p_s , e_s) are as follows.

Consumption Rule

The execution budget of the server is consumed at the rate of one per unit time whenever the server executes.

Replenishment Rule

The execution budget of the server is set to e_s at time instants kp_k, for $k = 0, 1, 2, \ldots$.

Any budget held by the server immediately before each replenishment time is lost.

As an example, let us look again at the system in Figure 7–2. Suppose that the task *(2.5, 0.5)* is a deferrable server. When it finds the aperiodic job queue empty at time 0, it suspends itself, with its execution budget preserved. When aperiodic job *A* arrives at 0.1, the deferrable server resumes and executes *A*. At time 0.6, its budget completely consumed, the server is suspended. It executes again at time 2.5 when its budget is replenished. When *A* completes at time 2.8, the aperiodic job queue becomes empty. The server is suspended, but it still has 0.2 unit of execution budget.

Figure 7–3 gives another example. Figure 7–3(a) shows that the deferrable server T_{DS} =*(3, 1)* has the highest priority. The periodic tasks T_1 = *(2.0, 3.5, 1.5)* and T_2 = *(6.5, 0.5)* and the server are scheduled rate-monotonically. Suppose that an aperiodic job *A* with execution time 1.7 arrives at time 2.8.

1. At time 0, the server is given 1 unit of budget. The budget stays at 1 until time 2.8.When *A* arrives, the deferrable server executes the job. Its budget decreases as it executes.
2. Immediately before the replenishment time 3.0, its budget is equal to 0.8. This 0.8 unit is lost at time 3.0, but the server acquires a new unit of budget. Hence, the server continues to execute.
3. At time 4.0, its budget is exhausted. The server is suspended, and the aperiodic job *A* waits.
4. At time 6.0, its budget replenished, the server resumes to execute *A*.
5. At time 6.5, job *A* completes. The server still has 0.5 unit of budget. Since no aperiodic job waits in the queue, the server suspends itself holding this budget.

Figure 7–3(b) shows the same periodic tasks and the deferrable server scheduled according to the EDF algorithm. At any time, the deadline of the server is equal to the next replenishment time.

1. At time 2.8, the deadline of the deferrable server is 3.0. Consequently, the deferrable server executes at the highest-priority beginning at this time.
2. At time 3.0, when the budget of the deferrable server is replenished, its deadline for consuming this new unit of budget is 6. Since the deadline of $J_{1,1}$ is sooner, this job has a higher priority. The deferrable server is preempted.
3. At time 3.7, $J_{1,1}$ completes. The deferrable server executes until time 4.7 when its budget is exhausted.
4. At time 6 when the server's budget is replenished, its deadline is 9, which is the same as the deadline of the job $J_{1,2}$. Hence, $J_{1,2}$ would have the same priority as the server.

We also use a ***background server***. This server is scheduled whenever the budget of the deferrable server has been exhausted and none of the periodic tasks is ready for execution. When the background server is scheduled, it also executes the aperiodic job at the head of the aperiodic job queue. With such a server, job *A* in Figure 7–3(a) would be executed from time 4.7 and completed by time 5.2, rather than 6.5, as shown in the figure.

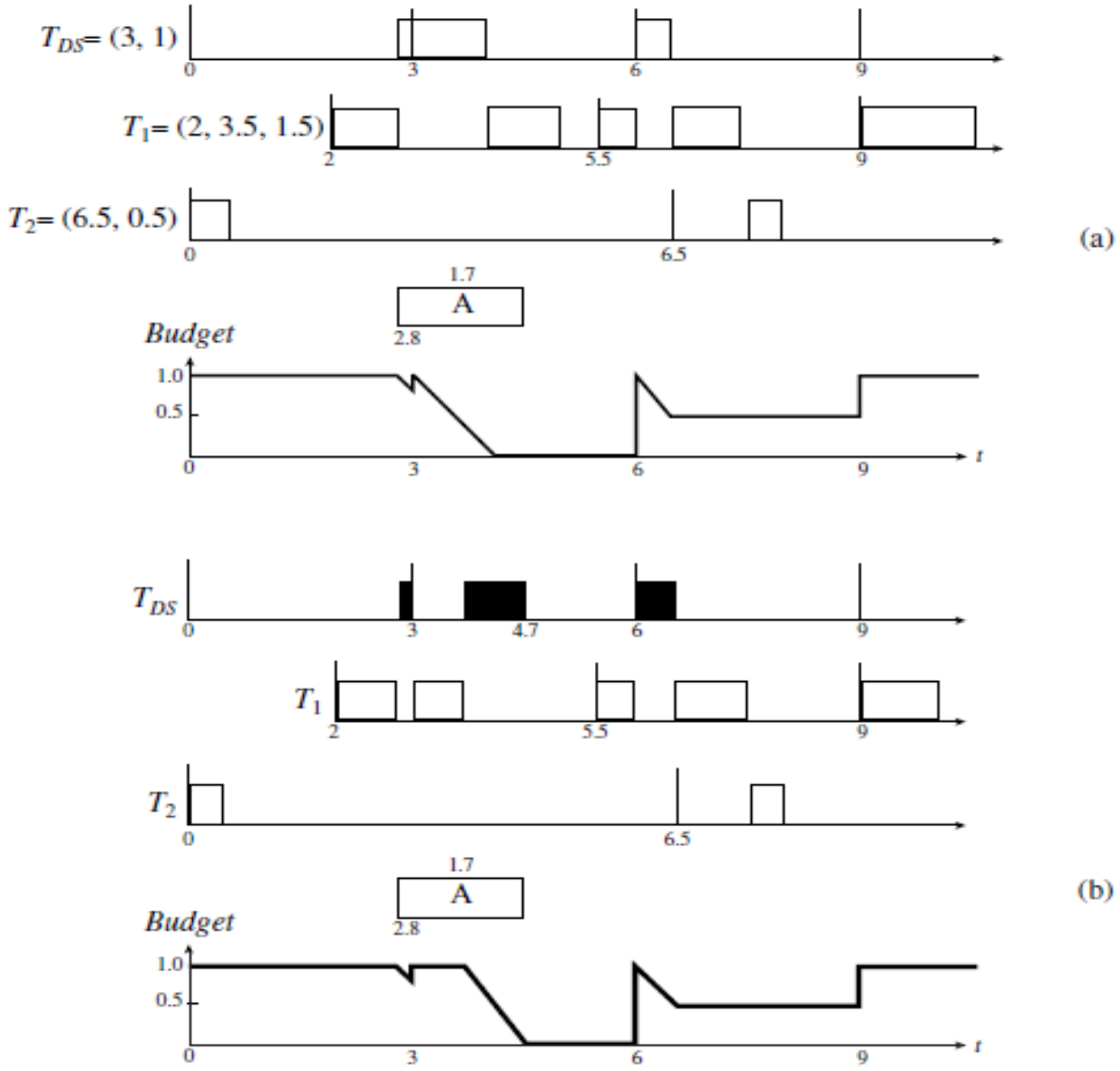


FIGURE 7–3 Example illustrating the operations of a deferrable server: (T_{DS} = (3, 1), T_1 = (2, 3.5, 1.5), and T_2 = (6.5, 0.5). (a) Rate-monotonic schedule. (b) EDF schedule.

7.2.2 Schedulability of Fixed-Priority Systems Containing Deferrable Server(s)

The schedule in Figure 7–3 may lead us to wonder whether we can get the same improved response by making the execution budget of the server bigger, for example, let it be 1.5 instead of 1.0. As it turns out, if we were to make the server budget bigger, the task *(*6.5*,* 0.5*)* would not be schedulable.

This fact is shown by Figure 7–4. Suppose that an aperiodic job with execution time 3.0 or more arrives to an empty aperiodic job queue at time t_0, 1 unit of time before the replenishment time of the server *(*3*,* 1*)*. At t_0, the server has 1 unit of budget. The server begins to execute from time t_0. Its budget is replenished at t_{0+1}, and it continues to execute as shown. Suppose that a job of each of the periodic tasks is released at time t_0. ***For this combination of release***

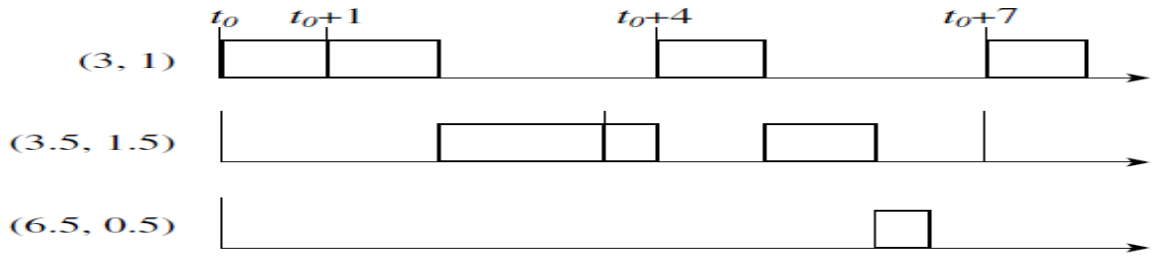


FIGURE 7–4 The factor limiting the budget of a deferrable server.

times, the task with period 3.5 would not complete in time if the budget of the server were bigger than 1.0. Since we cannot be sure that this combination of release times can never occur, the budget of the server must be chosen based the assumption that this combination can occur. ***This consideration limits the size of the server.***

LEMMA 7.1. In a fixed-priority system in which the relative deadline of every independent, preemptable periodic task is no greater than its period and there is a deferrable server (p_s , e_s) with the highest priority among all tasks, a critical instant of every periodic task T_i occurs at time t_0 when all the following are true.

1. One of its jobs $J_{i,c}$ is released at t_0.

2. A job in every higher-priority task is released at the same time.

3. The budget of the server is e_s at t_0, (one or more aperiodic jobs are released at t_0, and they keep the server backlogged hereafter).

4. The next replenishment time of the server is $t_0 + e_s$.

Figure 7–5 shows the segment of a fixed-priority schedule after a critical instant t_0. Task T_{DS} is the deferrable server (p_s , e_s). As always, the other tasks are indexed in decreasing order of their priorities. As far as the job $J_{i,c}$ that is released at t_0 is concerned, the demand for processor time by each of the higher-priority tasks in its feasible interval $(r_{i,c}, r_{i,c} + D_i]$ is the largest when (1) and (2) are true. When (3) and (4) are true, the amount of processor time consumed by the deferrable server in the feasible interval of $J_{i,c}$ is equal to $\boldsymbol{e_s + \lceil (D_i - e_s)/p_s \rceil e_s}$, and this amount is the largest possible.

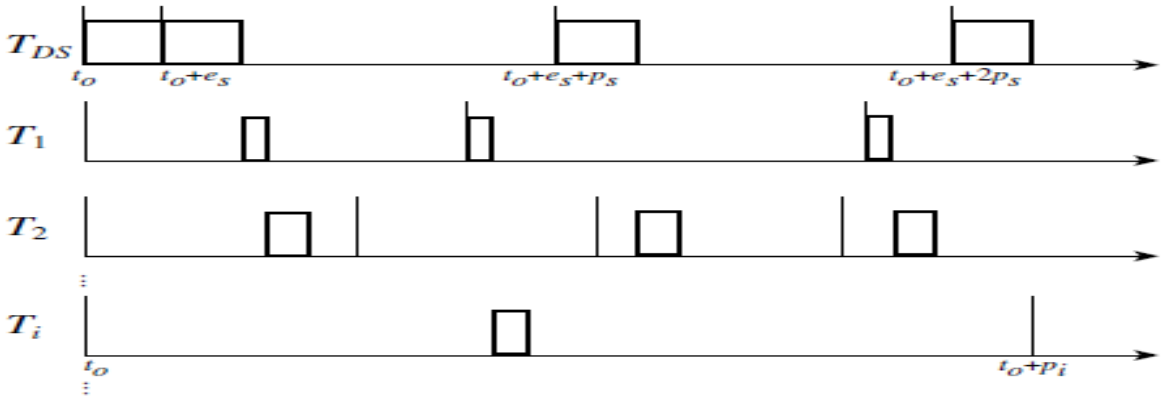


FIGURE 7–5 A segment of an arbitrary fixed-priority schedule after a critical instant of T_i.

Time-Demand Analysis Method. We can use the time-demand method to determine whether all the periodic tasks remain schedulable in the presence of a deferrable server (p_s , e_s). For a job $J_{i,c}$ in T_i that is released at a critical instant t_0, we add the maximum amount $\boldsymbol{e_s + \lceil (t - e_s)/p_s \rceil e_s}$ of processor time demanded by the server at time t units after t_0 into the right-hand side of Eq. (6.18). Hence, the time-demand function of the task T_i is given by

$$w_i(t) = e_i + b_i + e_s + \left\lceil \frac{t - e_s}{p_s} \right\rceil e_s + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k \qquad \text{for } 0 < t \le p_i \qquad (7.1)$$

when the deferrable server (p_s , e_s) has the highest priority.

Figure 7–6 shows the time-demand functions of T_1 and T_2 in the system shown in Figure 7–3. To determine whether T_2 is schedulable, we must check whether $w_i(t) \le t$ at 1 and 4,
which are the replenishment times of the server before the deadline 6.5 of the first job in T_2, in addition to 3.5, which is the period of T_1, and 6.5. As expected, both tasks are schedulable according to this test.

Schedulable Utilization. There is no known schedulable utilization that assures the schedulability of a fixed-priority system in which a deferrable server is scheduled at an arbitrary priority. The only exception is the special case when the period p_s of the deferrable server is shorter than the periods of all periodic tasks and the system is scheduled rate-monotonically.

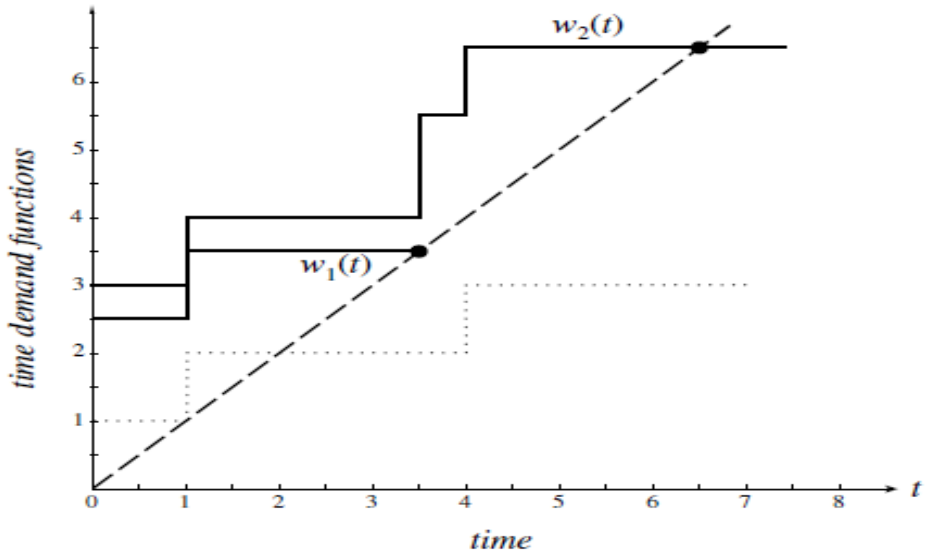


FIGURE 7–6 The time demand functions of $T_1 = (2, 3.5, 1.5)$ and $T_2 = (6.5, 0.5)$ with a deferrable server $(3, 1.0)$.

THEOREM 7.2. Consider a system of n independent, preemptable periodic tasks whose periods satisfy the inequalities $p_s < p_1 < p_2 < \cdots < p_n < 2p_s$ and $p_n > p_s + e_s$ and whose relative deadlines are equal to their respective periods. This system is schedulable rate-monotonically with a deferrable server (p_s , e_s) if their total utilization is less than or equal to

$$U_{RM/DS}(n) = (n-1)\left[\left(\frac{u_s + 2}{u_s + 1}\right)^{1/(n-1)} - 1\right]$$

where u_s is the utilization e_s/p_s of the server.

When the server's period is arbitrary, we can use the schedulable utilization $U_{RM}(n)$ given by Eq. (6.10) to determine whether each periodic task T_i is schedulable if the periodic tasks and the server are scheduled rate-monotonically and the relative deadlines of the periodic tasks are equal to their respective periods. To see how, we focus on the feasible interval $(r_{i,c}, r_{i,c} + D_i]$ of a job $J_{i,c}$ in any periodic task T_i that has a lower priority than the server.
As far as this job is concerned, the server behaves just like a periodic task (p_s , e_s), except that the server may execute for an additional e_s units of time in the feasible interval of the job. We can treat these e_s units of time as additional "blocking time" of the task T_i : T_i is surely schedulable if

$$U_i + u_s + \frac{e_s + b_i}{p_i} \le U_{RM}(i+1)$$

where U_i is the total utilization of the i highest-priority tasks in the system.

Deferrable Server with Arbitrary Fixed Priority. For some combinations of task and server parameters, the time demand of the server in a time interval of length t may never be as large as $\boldsymbol{e_s + \Gamma(t - e_s)/p_s\rceil e_s}$, the terms we included in the right-hand side of Eq. (7.1). However, these terms do give us an upper bound to the amount of time demanded by the server.

From this observation, we can conclude that in a system where the deferrable server is scheduled at an arbitrary priority, the time-demand function of a task T_i with a lower-priority than the server is bounded from above by the expression of $w_i(t)$ in Eq. (7.1). Using this upper bound, we make the time-demand analysis method a sufficient schedulability test for any fixed-priority system containing one deferrable server.

The time-demand function $w_i(t)$ of a periodic task T_i with a lower priority than m deferrable servers, each of which has period p_s,k and execution budget e_s,k, is given by

$$w_i(t) \le e_i + b_i + \sum_{k=1}^{m}\left(1 + \left\lceil \frac{t - e_{s,k}}{p_{s,k}} \right\rceil\right) e_{s,k} + \sum_{k=1}^{i-1}\left\lceil \frac{t}{p_k} \right\rceil e_k \qquad \text{for } 0 < t \le p_i \qquad (7.2)$$

7.2.3 Schedulability of Deadline-Driven Systems in the Presence of Deferrable Server

We now derive the schedulable utilization of a system of n independent periodic tasks that is scheduled according to the EDF algorithm together with a deferrable server of period p_s and execution budget es. Let t be the deadline of a job $J_{i,c}$ in some periodic task T_i and t_{-1} (< t) be the latest time instant at which the processor is either idle or is executing a job whose deadline is after t .We observe that if $J_{i,c}$ does not complete by t , the total amount $w_{DS}(t-t_{-1})$ of processor time consumed by the deferrable server in the interval $(t_{-1}, t\,]$ is at most equal to

$$e_s + \left\lfloor \frac{t - t_{-1} - e_s}{p_s} \right\rfloor e_s \qquad (7.3)$$

Figure 7–7 shows the condition under which the server consumes this amount of time.

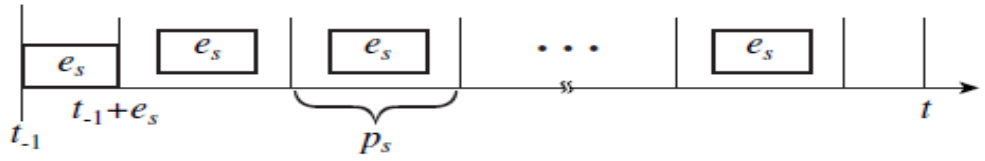


FIGURE 7–7 the condition under which the deferrable server consumes the most time.

1. At time t_{-1}, its budget is equal to e_s and the deadline for consuming the budget is $t_{-1} + e_s$.
2. One or more aperiodic jobs arrive at t_{-1}, and the aperiodic job queue is never empty hereafter, at least until t .
3. The server's deadline $t_{-1} + e_s$ is earlier than the deadlines of all the periodic jobs that are ready for execution in the interval $(t_{-1}, t_{-1} + e_s\,]$.

The total amount of the time consumed by the server after $t_{-1} + e_s$ is given by the second term in Eq. (7.3). Since $\lfloor x \rfloor \le x$ for all $x \ge 0$ and $u_s = e_s/p_s$, $w_{DS}(t - t_{-1})$ must satisfy the following inequality if the job $J_{i,c}$ does not complete by its deadline at t :

$$w_{DS}(t - t_{-1}) \le e_s + \frac{t - t_{-1} - e_s}{p_s} e_s = u_s(t - t_{-1} + p_s - e_s) \qquad (7.4)$$

THEOREM 7.3. A periodic task T_i in a system of n independent, preemptive periodic tasks is schedulable with a deferrable server with period p_s , execution budget e_s, and utilization u_s , according to the EDF algorithm if

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} + u_s \left(1 + \frac{p_s - e_s}{D_i}\right) \le 1 \qquad (7.5)$$

Proof. We suppose that a job misses its deadline at t and consider the interval $(t_{-1}, t]$. The interval begins at the latest time instant t_{-1} at which time the processor is either idle or is executing a job whose deadline is after t . The total amount of processor time used by each periodic task T_k in the time interval is bounded from above by $e_k(t - t_{-1})/p_k$.

The fact that a job misses its deadline at t tells us that the total amount of time in this time interval used by all the periodic jobs with deadlines at or before t and the deferrable server exceeds the length of this interval. In other words,

$$t - t_{-1} \le \sum_{k=1}^{n} \frac{e_k}{p_k}(t - t_{-1}) + u_s(t - t_{-1} + p_s - e_s)$$

Dividing both sides of this inequality by $t - t_{-1}$, we get Eq. (7.5) for the case of $D_k \ge p_k$.

7.3 SPORADIC SERVERS

We have just seen that a deferrable server may delay lower-priority tasks for more time than a period task with the same period and execution time. This section describes a class of bandwidth-preserving servers, called ***sporadic servers*** , that are designed to improve over a deferrable server in this respect. The consumption and replenishment rules of sporadic server algorithms ensure that each sporadic server with period p_s and budget e_s never demands more processor time than the periodic task (p_s, e_s) in any time interval. Consequently, we can treat the sporadic server exactly like the periodic task (p_s , e_s) when we check for the schedulability of the system. A system of periodic tasks containing a sporadic server may be schedulable while the same system containing a deferrable server with the same parameters is not.

7.3.1 Sporadic Server in Fixed-Priority Systems

We assume that there is only one sporadic server in a fixed-priority system **T** of n independent, preemptable periodic tasks. The server has an arbitrary priority π_s . We use $\mathbf{T}_H$ to denote the subset of periodic tasks that have higher priorities than the server. We say that the system **T** of periodic tasks idles when no job in **T** (or $\mathbf{T}_H$) is ready for execution; **T** (or $\mathbf{T}_H$) is busy otherwise. By definition, the higher-priority subsystem remains busy in any busy interval of $\mathbf{T}_H$. Finally, a ***server busy interval*** is a time interval which begins when an aperiodic job arrives at an empty aperiodic job queue and ends when the queue becomes empty again.

We state below the consumption and replenishment rules that define a simple sporadic server. In the statement, we use the following notations.

- t_r denotes the latest (actual) replenishment time.
- t_f denotes the first instant after t_r at which the server begins to execute.
- t_e denotes the latest ***effective replenishment time***.
- At any time t , *BEGIN* is the beginning instant of the earliest busy interval among the latest contiguous sequence of busy intervals of the higher-priority subsystem $\mathbf{T}_H$ that started before t .
- *END* is the end of the latest busy interval in the above defined sequence if this interval ends before t and equal to

infinity if the interval ends after t .

The scheduler sets t_r to the current time each time it replenishes the server's execution budget. When the server first begins to execute after a replenishment, the scheduler determines the latest effective replenishment time t_e based on the history of the system and sets the next replenishment time to $t_e + p_s$.

Simple Sporadic Server. In its simpliest form, a sporadic server is governed by the following consumption and replenishment rules. We call such a server a ***simple sporadic server***. A way to implement the server is to have the scheduler monitor the busy intervals of $\mathbf{T}_H$ and maintain information on *BEGIN* and *END*.

• ***Consumption Rules of Simple Fixed-Priority Sporadic Server*:** At any time t after t_r , the server's execution budget is consumed at the rate of 1 per unit time until the budget is exhausted when either one of the following two conditions is true. When these conditions are not true, the server holds its budget.
C1 The server is executing.
C2 The server has executed since t_r and $END < t$.

• ***Replenishment Rules of Simple Fixed-Priority Sporadic Server*:**
R1 Initially when the system begins execution and each time when the budget is replenished, the execution budget = e_s, and t_r = the current time.
R2 At time t_f , if $END = t_f$, $t_e = \max(t_r , BEGIN)$. If $END < t_f$, $t_e = t_f$. The next replenishment time is set at $t_e + p_s$.
R3 The next replenishment occurs at the next replenishment time, except under the following conditions.

Under these conditions, replenishment is done at times stated below.
(a) If the next replenishment time $t_e + p_s$ is earlier than t_f , the budget is replenished as soon as it is exhausted.
(b) If the system **T** becomes idle before the next replenishment time $t_e + p_s$ and becomes busy again at t_b, the budget is replenished at $\min(t_e + p_s , t_b)$.
Figure 7–8 gives an illustrative example. Initially the budget of the server *(*5*,* 1.5*)* is 1.5. It is scheduled rate-monotonically with three periodic tasks: T_1 = *(*3*,* 0.5*)*, T_2 = *(*4*,* 1.0*)*, and T_3 = *(*19*,* 4.5*)*. They are schedulable even when the aperiodic job queue is busy all the time.

1. From time 0 to 3, the aperiodic job queue is empty and the server is suspended. Since it has not executed, its budget stays at 1.5. At time 3, the aperiodic job A_1 with execution time 1.0 arrives; the server becomes ready. Since the higher-priority task *(*3*,* 0.5*)* has a job ready for execution, the server and the aperiodic job wait.
2. The server does not begin to execute until time 3.5. At the time, t_r is 0, *BEGIN* is equal to 3, and *END* is equal to 3.5. According to rule R2, the effective replenishment time t_e is equal to max*(*0*,* 3.0*)* = 3, and the next replenishment time is set at 8.
3. The server executes until time 4; while it executes, its budget decreases with time.
4. At time 4, the server is preempted by T_2.While it is preempted, it holds on to its budget.
5. After the server resumes execution at 5, its budget is consumed until exhaustion because first it executes (C1) and then, when it is suspended again, T_1 and T_2 are idle.
6. When the aperiodic job A_2 arrives at time 7, the budget of the server is exhausted; the job waits in the queue.
7. At time 8, its budget replenished (R3), the server is ready for execution again.
8. At time 9.5, the server begins to execute for the first time since 8. t_e is equal to the latest replenishment time 8.

Hence the next replenishment time is 13. The server executes until its budget is exhausted at 11; it is suspended and waits for the next replenishment time. In the meantime, A_2 waits in the queue.

9. Its budget replenished at time 13, the server is again scheduled and begins to execute at time 13.5. This time, the next replenishment time is set at 18.

However at 13.5, the periodic task system **T** becomes idle. Rather than 18, the budget is replenished at 15, when a new busy interval of **T** begins, according to rule R3b.

10. The behavior of the later segment also obeys the above stated rules. In particular, rule R3b allows the server budget to be replenished at 19.

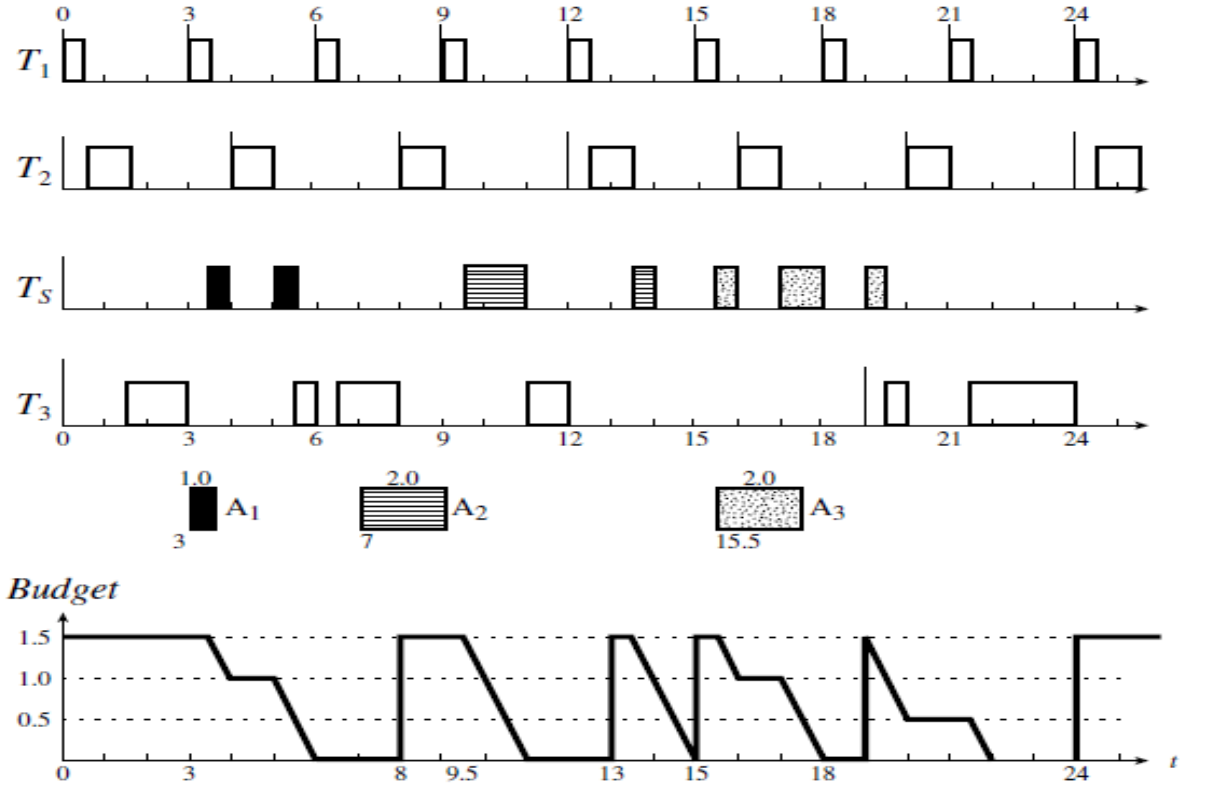


FIGURE 7–8 Example illustrating the operations of a simple sporadic server: $T_1 = (3, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$, $T_s = (5, 1.5)$.

***Informal Proof of Correctness of the Simple Sporadic Server.** We now explain why a server following the above stated rules emulates the periodic task $T_s = (p_s , e_s)$; therefore, when checking for the schedulability of the periodic tasks, we can simply treat the server as the periodic task T_s .

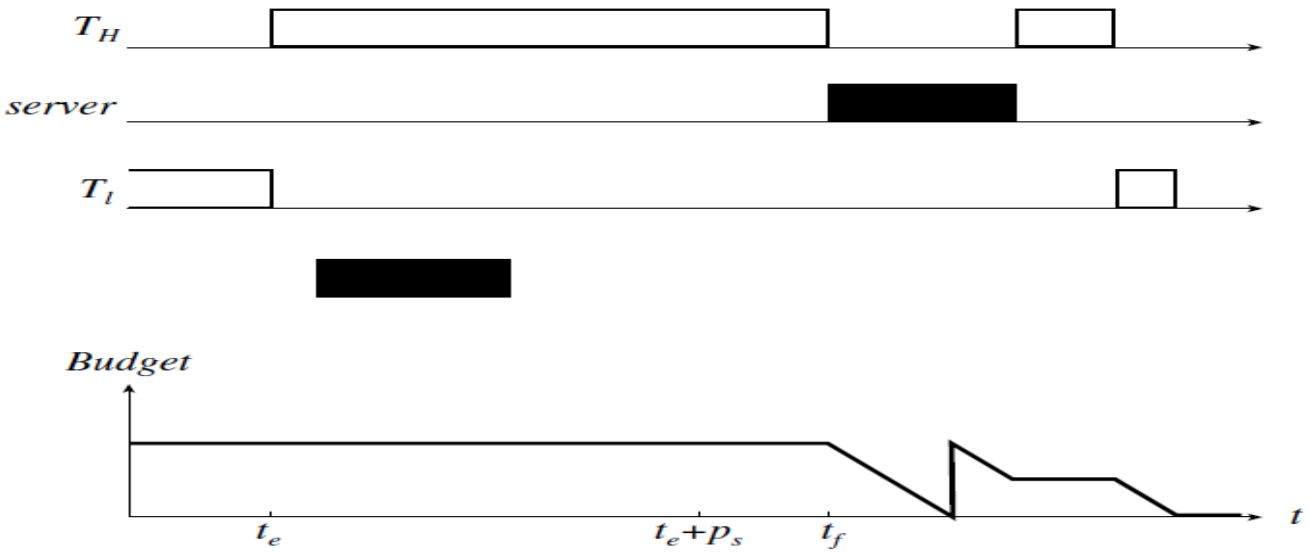


FIGURE 7–9 A situation where rule R3a applies.

The consumption rule C1 ensures that each server job never executes for more time than its execution budget e_s . C2 applies when the server becomes idle while it still has budget. Because of these two rules, each server job never executes at times when the corresponding job of the periodic task T_s does not.

To show the correctness of the replenishment rules R2 and R3a, we note that the next replenishment time is always set at the p_s time units after the effective release-time t_e of the current server job and the next release-time is never earlier than the next replenishment time.

Hence according to these two rules, consecutive replenishments occurs at least p_s units apart. Specifically, rule R2 is designed to make the effective replenishment time as soon as possible without making the server behave differently from a periodic task.

Rule R3a applies when the current server job has to wait for more than p_s units of time before its execution can begin. This rule works correctly if at the time when the server parameters were chosen and schedulability of the periodic tasks were determined. When rule R3b applies, the server may behave differently from the periodic task T_s . This rule is applicable only when a busy interval of the periodic task system **T** ends.

7.3.2 Enhancements of Fixed-Priority Sporadic Server

Sporadic/Background Server. The example in Figure 7–8 tells us that rule R3b of the simple fixed-priority sporadic server is overly conservative. At time 18.5, the periodic system becomes idle. The aperiodic job A_3 remains incomplete, but the server budget is exhausted. According to this rule, the budget is not replenished until the job $J_{3,2}$ is released at 19. While A_3 waits, the processor idles. It is clear that the schedulability of the periodic system will not be adversely affected if we replenish the server budget at 18.5 and let the server execute until 19 with its budget undiminished.

We call a sporadic server that claims the background time a ***sporadic/background server***; in essence, it is a combination of a sporadic server and a background server and is defined by the following rules.

• ***Consumption rules of simple sporadic/background servers*** are the same as the rules of simple sporadic servers except when the period task system is idle. As long as the periodic task system is idle, the execution budget of the server stays at e_s .
• ***Replenishment rules of simple sporadic/background servers*** are same as those of the simple sporadic server except R3b. The budget of a sporadic/background server is replenished at the beginning of each idle interval of the periodic task system. t_r is set at the end of the idle interval.

As an example, suppose that a system has two aperiodic tasks. The jobs in one task are released whenever the operator issues a command, and their execution times are small. We use a sporadic server to execute these jobs. In order to make the system responsive to the operator, we choose to make the server period 100 milliseconds and execution budget sufficiently large to complete a job (at least most of time). Each job of the other aperiodic task is released to process a message. The messages arrive infrequently, say one a minute on the average.

The execution times of jobs in this task vary widely because the messages have widely differing lengths. We may want to give all the background time to the message-processing server. This can be done by making the command-processing server a simple sporadic server and the message-processing server a sporadic/background server.

Cumulative Replenishment. A reason for the simplicity of simple sporadic servers is that all the budget a server has at any time was replenished at the same time. A simple server is not allowed to cumulate its execution budget from replenishment to replenishment. This simplifies the consumption and replenishment rules.

A way to give the server more budget is to let the server keep any budget that remains unconsumed. Hence the budget of a server may exceed e_s . As a result of this change in replenishment rule R1, the server emulates a periodic task in which some jobs do not complete before the subsequent jobs in the task are released.

Figure 7–10 gives an example. The server with period 15 and execution budget 6 cannot consume its budget at time 15 when its budget is replenished. This example illustrates that it is safe to increment its budget by 6 units. By doing so, the server emulates the periodic task *(*15*,* 6*)* whose first job cannot complete within one period. Due to this, the server completes the aperiodic job A_2 by time 27. In contrast, a simple sporadic server cannot complete the job until time 36.

At any time t after the latest replenishment time t_r , it is still safe to consume the new budget according to C1 and C2. However, we cannot use rule C2 to govern the consumption of the old budget. After t_r , the old budget should be consumed at the rate of 1 per unit time when the server is suspended and the higher-priority subsystem $\mathbf{T}_H$ is idle independent of whether the server has executed since t_r .

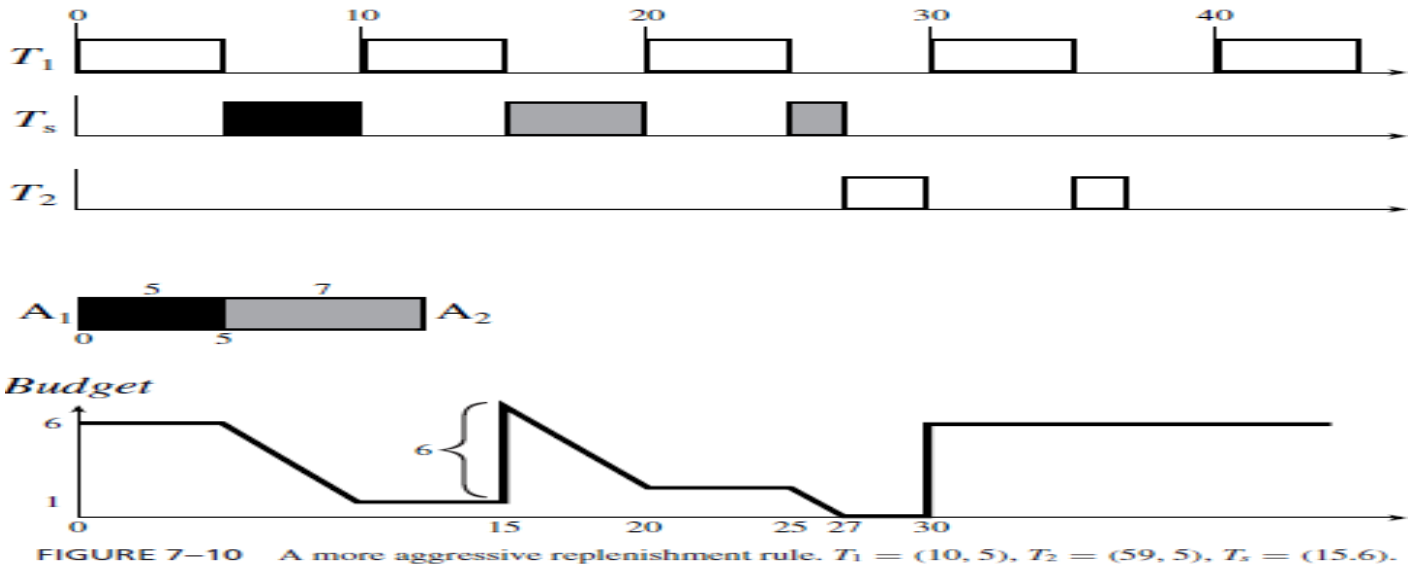


FIGURE 7–10 A more aggressive replenishment rule. $T_1 = (10, 5)$, $T_2 = (59, 5)$, $T_s = (15.6)$.

We can further improve the replenishment rule R3a as follows:
Replenish the budget at time $t_r + p_s$ whenever the higher-priority subsystem $\mathbf{T}_H$ has been busy throughout the interval $(t_r , t_r + p_s]$. This additional change in replenishment rules in turn makes it necessary to treat the budget that is not consumed in this interval with care.

SpSL Sporadic Servers. Specifically, a Sprunt, Sha, and Lehoczky (SpSL) sporadic server preserves unconsumed chunks of budget whenever possible and replenishes the consumed chunks as soon as possible. By doing so, a SpSL server with period p_s and execution budget e_s emulates several periodic tasks with the same period and total execution time equal to e_s .

To explain, we return to the example in Figure 7–8. The figure shows that at time 5.5 when the simple sporadic server is suspended, the server has already consumed a 1-unit chunk of budget. Its remaining 0.5-unit chunk of budget is consumed as the lower-priority task T_3 executes. If an aperiodic job A_4 with 0.5 unit of execution time were to arrive at 6.5, the server would have to wait until time 9.5 to execute the job. In contrast, a SpSL server holds on to the remaining chunk of 0.5 unit of budget as T_3 executes. Figure 7–11 illustrates the operations of a SpSL server.

1. Suppose that at time 6.5, A_4 indeed arrives. The SpSL server can execute the job immediately, and in the process, consumes its remaining chunk of budget.
2. At time 8, rather than the entire budget of 1.5 units, the SpSL server replenishes only the chunk of 1.0 unit which was consumed from time 3.5 to 5.5. This 1.0 unit is treated as one chunk because the execution of the server was interrupted only by a higher-priority task during the time interval when this chunk was consumed.

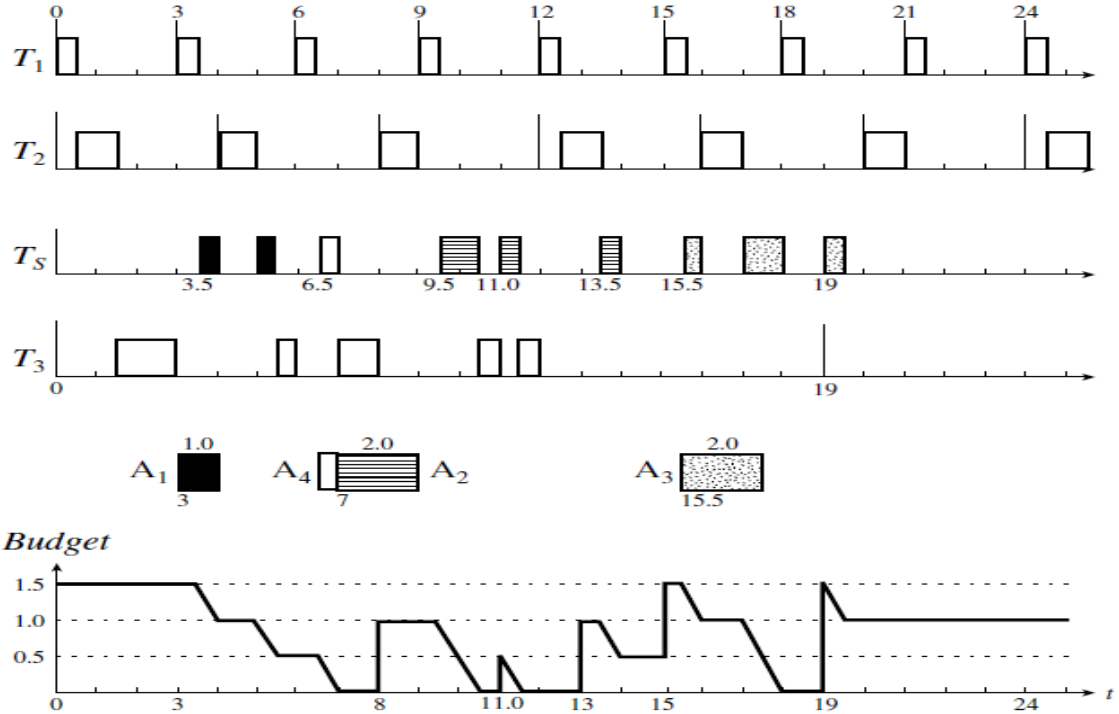


FIGURE 7–11 Example illustrating SpSL sporadic servers: $T_1 = (3, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$, $T_s = (5, 1.5)$.

3. At time 11.0, 5 units of time from the effective replenishment time of the second chunk, the 0.5 unit of budget is replenished.

4. Similarly, at time 13, only the first chunk is replenished.

5. Later at 13.5, when the higher-priority tasks are no longer busy, the server executes and completes *A*2. At this time, it still has 0.5 units of budget, and this chunk is preserved.

By now, the server has three different chunks of budget: the chunk that was replenished at time 11.0 and consumed at 11.5, the chunk that was replenished at 13 and consumed by 14.0, and the leftover chunk at 14.0.

6. As it is, the periodic task system becomes idle at 14.0. All the chunks can be replenished at time 15 following rule R3b of the simple sporadic server, and the server has only one 1.5-unit chunk of budget again.

This example illustrates the advantage of SpSL servers over simple sporadic servers. Conceptually, a SpSL server emulates several periodic tasks, one per chunk. As time progresses and its budget breaks off into more and more chunks. The additional cost of SpSL servers over simple servers arises due to the need of keeping track of the consumption and replenishment of different chunks of budget.

Rules of SpSL Server

• Breaking of Execution Budget into Chunks

B1 Initially, the budget = e_s and t_r = 0. There is only one chunk of budget.

B2 Whenever the server is suspended, the last chunk of budget being consumed just before suspension, if not exhausted, is broken up into two chunks: The first chunk is the portion that was consumed during the last server busy interval, and the second chunk is the remaining portion. The first chunk inherits the next replenishment time of the original chunk. The second chunk inherits the last replenishment time of the original chunk.

• ***Consumption Rules:***

C1 The server consumes the chunks of budget in order of their last replenishment times.

C2 The server consumes its budget only when it executes.

• ***Replenishment Rules:*** The next replenishment time of each chunk of budget is set according to rules R2 and R3 of the simple sporadic server. The chunks are consolidated into one whenever they are replenished at the same time.

The overhead of the SpSL server over the simple version consists of the time and space required to maintain the last and the next replenishment times of individual chunks of the server budget. In the worst case, this can be as high as *O(N)*, where *N* is the number of jobs in a hyperperiod of the periodic tasks.

***Priority Exchanges.** In addition to allowing the budget to be divided up into chunks as a SpSL server does, there is another way to preserve the server budget. To see how, we begin by reexamining rule C2 of the simple sporadic server. According to this rule, once the server has executed since a replenishment, it consumes its budget except during the time interval(s) when it is preempted.

The schedule segments in Figure 7–12 illustrate what we mean. T_l is a lower-priority task. The schedule in part (a) illustrates the situation when the aperiodic job queue remains nonempty after the server begins to execute at time t_f. The server continues to execute until its budget is exhausted at time $t_f + e_s$. The lower-priority task T_l must wait until this time to execute and completes later at time t_l. The schedule in part (b) illustrates the situation that the server busy interval ends and the server becomes idle and, therefore, suspended at time $t_f + \varepsilon$ before its budget is exhausted.

A job in the lower priority task T_l begins to execute at this time. Because T_l is able to use the $e_s - \varepsilon$ units of time which would be given to the server if it were not suspended, T_l can complete sooner. This means that $e_s - \varepsilon$ units of time before t_l is no longer needed by the lower-priority task; they can be given to the server as budget.

Figure 7–12(b) shows that an aperiodic job arrives later at time t_0, and it can be executed by the server if the server has the traded budget, while a simple sporadic server has no budget at this time. In general, we may have the situation where (1) the server becomes idle and is therefore suspended while it still has budget and (2) some lower priority tasks $T_l, T_{l+1}, \ldots, T_{l+k}$ execute in the time interval where the server would be scheduled if it were not suspended.

The lower-priority tasks execute at the server's priority in this interval. Let x_j denote the amount of time the lower-priority task T_{l+j} executes in this interval. The server is allowed to keep x_j units of budget. When the server later uses this budget, it will be scheduled at the priority of the task T_{l+j}. Trading time and priorities among the server and the lower-priority tasks in this manner is exactly what a ***priority-exchange server*** does. The high overhead of priority exchange makes this type of servers impractical.

7.3.3 Simple Sporadic Servers in Deadline-Driven Systems

When the periodic tasks are scheduled on the EDF basis, the priorities of the tasks vary as their jobs are released and completed. Consequently, the membership of the subset of tasks with higher priorities than the server varies with time. Some of the rules of simple sporadic servers stated earlier for fixed-priority systems must be modified for this reason. The rationales behind the modified rules remain the same: a simple sporadic server of period p_s and budget e_s following the rules behave like a periodic task (p_s, e_s).

• ***Consumption Rules of Simple Deadline-Driven Sporadic Server*:** The server's execution budget is consumed at the rate of one per unit time until the budget is exhausted when either one of the following two conditions is true. When these conditions are not true, the server holds its budget.

C1 The server is executing.

C2 The server deadline *d* is defined, the server is idle, and there is no job with a deadline before *d* ready for execution.

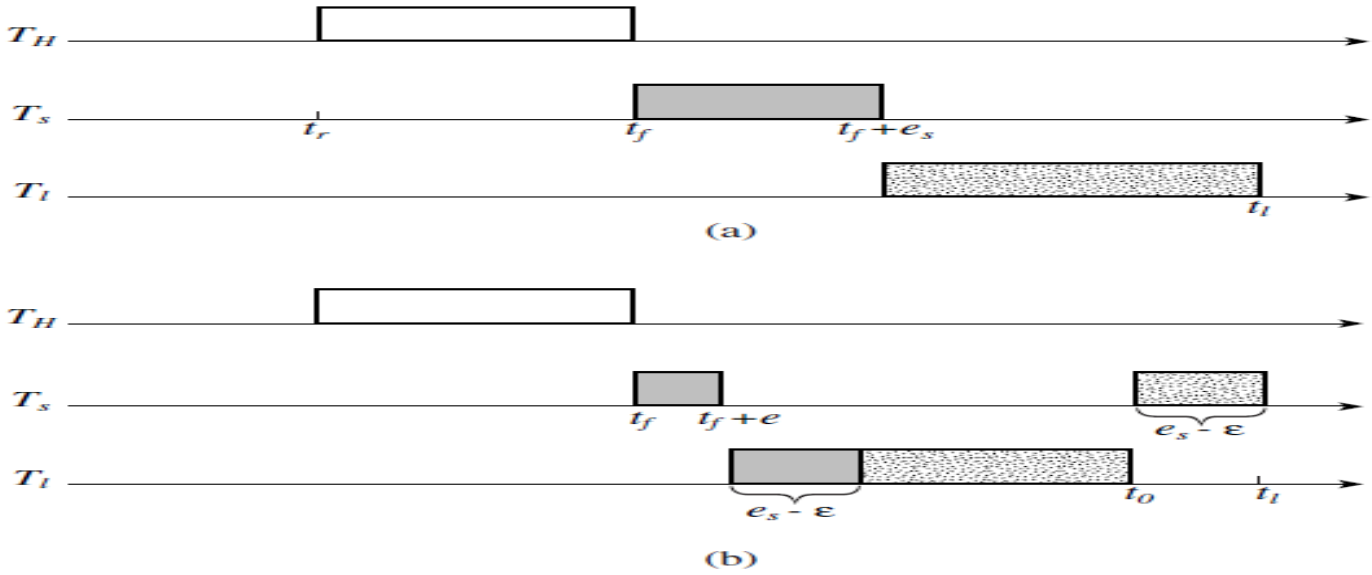


FIGURE 7–12 Priority exchange between the server and a lower-priority task.

• ***Replenishment Rules of Simple Deadline-Driven Sporadic Server*:**

R1 Initially and at each replenishment time, t_r is the current time, and the budget = e_s . Initially, t_e and the server deadline *d* are undefined.

R2 Whenever t_e is defined, $d = t_e + p_s$, and the next replenishment time is $t_e + p_s$.

Otherwise, when t_e is undefined, *d* remains undefined. t_e is determined (defined) as follows:

(a) At time *t* when an aperiodic job arrives at an empty aperiodic job queue, the value of t_e is determined based on the history of the system before *t* as follows:

i. If only jobs with deadlines earlier than $t_r + p_s$ have executed throughout the interval (t_r , t), $t_e = t_r$.

ii. If some job with deadline after $t_r + p_s$ has executed in the interval (t_r , t), $t_e = t$.

(b) At replenishment time t_r ,

i. if the server is backlogged, $t_e = t_r$, and

ii. if the server is idle, *te* and *d* become undefined.

R3 The next replenishment occurs at the next replenishment time, except under the following conditions. Under these conditions, the next replenishment is done at times stated below.

(a) If the next replenishment time t_e+p_s is earlier than the time *t* when the server first becomes backlogged since *tr* , the budget is replenished as soon as it is exhausted.

(b) The budget is replenished at the end of each idle interval of the periodic task system **T**.

7.4 CONSTANT UTILIZATION, TOTAL BANDWIDTH, AND WEIGHTED FAIR-QUEUEING SERVERS

The three bandwidth preserving server algorithms that offer a simple way to schedule aperiodic jobs in deadline-driven systems are constant utilization, total bandwidth, and weighted fair-queueing algorithms. These algorithms belong to a class of algorithms that more or less emulate the ***Generalized Processor Sharing (GPS) algorithm.***

GPS, sometimes called fluid-flow processor sharing, is an idealized weighted roundrobin algorithm; it gives each backlogged server in each round an infinitesmally small time slice of length proportional to the server size.

7.4.1 Schedulability of Sporadic Jobs in Deadline-Driven Systems

The ***density*** of a sporadic job J_i that has release time r_i , maximum execution time e_i and deadline d_i is the ratio $e_i /(d_i - r_i)$. A sporadic job is said to be ***active*** in its feasible interval $(r_i , d_i]$; it is not active outside of this interval.

THEOREM 7.4. A system of independent, preemptable sporadic jobs is schedulable according to the EDF algorithm if the total density of all active jobs in the system is no greater than 1 at all times.

Proof. We prove the theorem by contradiction. Suppose that a job misses its deadline at time t , and there is no missed deadline before t. Let t_{-1} be the latest time instant before t at which either the system idles or some job with a deadline after t executes.

Suppose that k jobs execute in the time interval $(t_{-1}, t\ ]$.We call these jobs $J_1, J_2, \ldots, J_k$ and order them in increasing order of their deadlines. J_k is the job that misses its deadline at t . Because the processor remains busy in $(t_{-1}, t\ ]$ executing jobs of equal or higher priorities than J_k and J_k misses its deadline at time t , we must have

$\sum_{i=1}^{k} e_i > t - t_{-1}$.

We let the number of job releases and completions during the time interval (t_{-1}, t) be l, and t_i be the time instant when the ith such event occurs. In terms of this notation, $t_{-1} = t_1$ and, for the sake of convenience, we also use t_{l+1} to denote t. These time instants partition the interval $(t_{-1}, t\ ]$ into l disjoint subintervals, $(t_1, t_2], (t_3, t_4], \ldots, (t_l, t_{l+1}]$. The active jobs in the system and their total density remain unchanged in each of these subintervals.

Let $\mathbf{X}_i$ denote the subset containing all the jobs that are active during the subinterval $(t_i, t_{i+1}]$ for $1 \le i \le l$ and Δ_i denote the total density of the jobs in $\mathbf{X}_i$.

The total time demanded by all the jobs that execute in the time interval $(t-1, t\ ]$ is $\sum_{i=1}^{k} e_i$. We can rewrite the sum as

$$\sum_{i=1}^{k} \frac{e_i}{d_i - r_i}(d_i - r_i) = \sum_{j=1}^{l}(t_{j+1} - t_j) \sum_{J_k \in \mathbf{X}_j} \frac{e_k}{d_k - r_k} = \sum_{j=1}^{l} \Delta_j (t_{j+1} - t_j)$$

Since $\Delta_j \le 1$ for all $j = 1, 2, \ldots, l-1$, we have

$$\sum_{i=1}^{k} e_i \le \sum_{j=1}^{l}(t_{j+1} - t_j) = t_{l+1} - t_1 = t - t_{-1}$$

This leads to a contradiction.

A Sporadic task S_i is a stream of sporadic jobs. Let $S_{i,j}$ denote the j th job in the task S_i (i.e., the release time of $S_{i,j}$ is later than the release times of $S_{i,1}, S_{i,2}, \ldots, S_{i,j-1}$).

Let $e_{i,j}$ denote the execution time of $S_{i,j}$, and $p_{i,j}$ denote the length of time between the release times of $S_{i,j}$ and $S_{i,j+1}$. At the risk of abusing the term, we call $p_{i,j}$ the *period* of the sporadic job $S_{i,j}$ and the ratio $e_{i,j}/p_{i,j}$ the ***instantaneous utilization*** of the job. The ***instantaneous utilization*** ($\tilde{u}_i$) of a sporadic task is the maximum of the instantaneous utilizations of all the jobs in this task (i.e., $\tilde{u}_i = \max_j (e_{i,j}/p_{i,j})$).

In a system of n sporadic tasks whose total instantaneous utilization is equal to or less than one, the total density of all active jobs is equal to or less than 1 at all times. Consequently, the following sufficient schedulability condition of sporadic tasks scheduled according to the EDF algorithm follows straightforwardly from Theorem 7.4.

COROLLARY 7.5. A system of n independent, preemptable sporadic tasks, which is such that the relative deadline of every job is equal to its period, is schedulable on a processor according to the EDF algorithm if the total instantaneous utilization (i.e., $\sum_{i=1}^{n} \tilde{u}_i$), is equal to or less than 1.

Because the utilization $u_i = \max_j (e_{i,j})/\min_j (p_{i,j})$ of any task S_i is always larger than its instantaneous utilization $\tilde{u}_i = \max_j (e_{i,j}/p_{i,j})$, we have the following corollary.

COROLLARY 7.6. A system of independent, preemptable periodic and sporadic tasks, which is such that the relative deadline of every job is equal to its period, is schedulable on a processor according to the EDF algorithm if the sum of the total utilization of the periodic tasks and the total instantaneous utilization of the sporadic tasks is equal to or less than 1.

7.4.2 Constant Utilization Server Algorithm

We now return our attention to the problem of scheduling aperiodic jobs amid periodic tasks in a deadline-driven system. For the purpose of executing aperiodic jobs, there is a **basic *constant utilization server***. The server is defined by its *size*, which is its instantaneous utilization $\tilde{u}_s$; this fraction of processor time is reserved for the execution of aperiodic jobs.

As with deferrable servers, the deadline d of a constant utilization server is always defined. It also has an execution budget which is replenished according to the replenishment rules described below. The server is eligible and ready for execution only when its budget is nonzero. While a sporadic server emulates a periodic task, a constant utilization server emulates a sporadic task with a constant instantaneous utilization, and hence its name.

Consumption and Replenishment Rules. The consumption rule of a constant utilization server, as well as that of a total bandwidth or weighted fair-queueing server, is quite simple. *A **server consumes its budget only when it executes**.*

The budget of a basic constant utilization server is replenished and its deadline set according to the following rules. In the description of the rules, $\tilde{u}_s$ is the size of the server, e_s is its budget, and d is its deadline. t denotes the current time, and e denotes the execution time of the job at the head the aperiodic job queue. The job at the head of the queue is removed when it completes. The rules assume that the execution time e of each aperiodic job becomes known when the job arrives.

Replenishment Rules of a Constant Utilization Server of Size $\tilde{u}_s$

R1 Initially, $e_s = 0$, and $d = 0$.

R2 When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue,
(a) if $t < d$, do nothing;
(b) if $t \geq d$, $d = t + e/\tilde{u}_s$, and $e_s = e$.

R3 At the deadline d of the server,
(a) if the server is backlogged, set the server deadline to $d + e/\tilde{u}_s$ and $e_s = e$;
(b) if the server is idle, do nothing.

Figure 7–13 illustrates how a constant utilization server works. This system of periodic tasks and aperiodic jobs is essentially the same as the system in Figure 7–8. The only difference between them is that in Figure 7–13, aperiodic job A_2 arrives at time 6.9 instead of 7.0. The size of the constant utilization server is 0.25, slightly smaller than the size of the sporadic server in Figure 7–8.

1. Before time 3.0, the budget of the server is 0. Its deadline is 0. The server does not affect other tasks because it is suspended.
2. At time 3, A_1 arrives. The budget of the server is set to 1.0, the execution time of A_1, and its deadline is 3+1.0/0.25 = 7 according to R2b. The server is ready for execution.

It completes $A1$ at time 4.5.

3. When A_2 arrives at time 6.9, the deadline of the server is later than the current time. According to R2a, nothing is done except putting A_2 in the aperiodic job queue.
4. At the next deadline of the server at 7, the aperiodic job queue is checked and A_2 is found waiting. The budget of

the server is replenished to 2.0, the execution time of A_2, and its deadline is 7 + 2.0/0.25 = 15.

The server is scheduled and executes at time 7, is preempted by T_2 at time 8, resumes execution at 9.5 and completes *A*2 at time 10.5.

5. At time 15, the aperiodic job queue is found empty. Nothing is done.

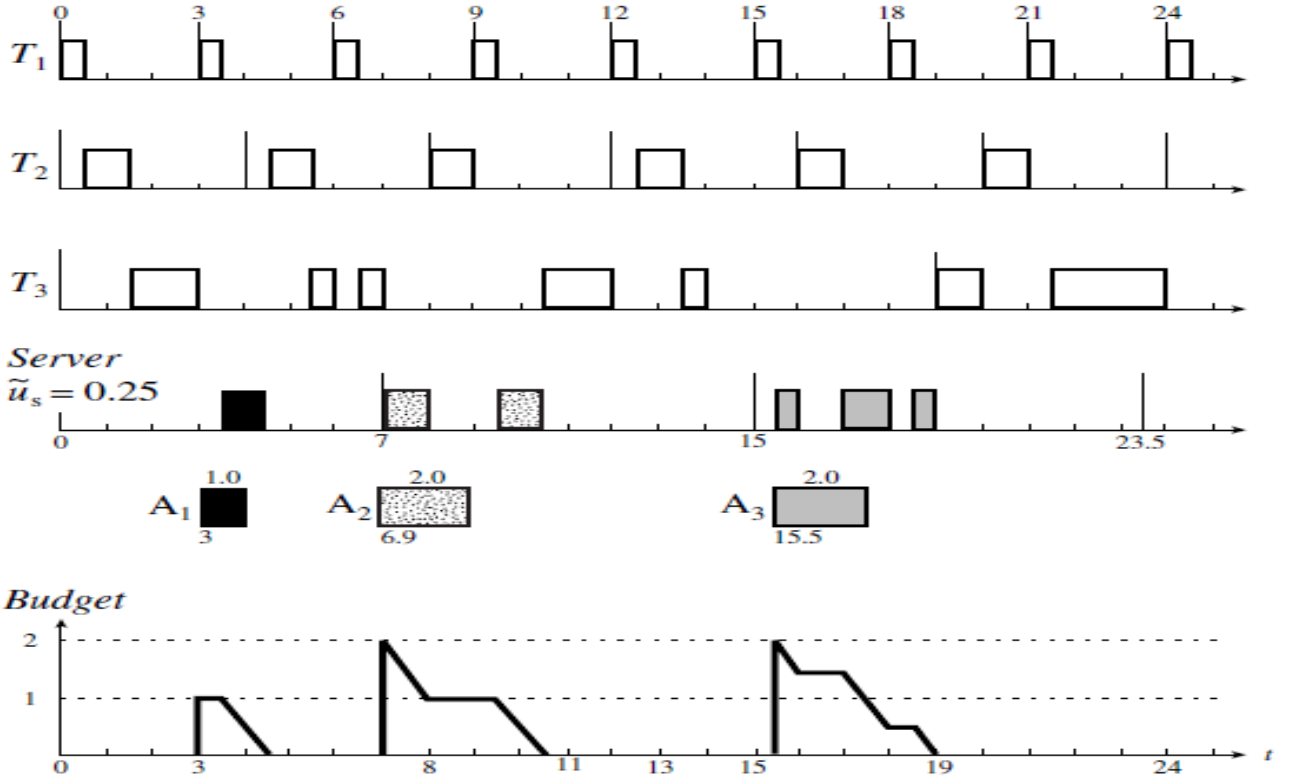


FIGURE 7–13 Example illustrating the operations of constant utilization server: $T_1 = (4, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$.

6. At time 15.5, A_3 arrives. At the time, the deadline of the server is 15. Hence according to rule R2b, its deadline is set at 23.5, and the server budget is set to 2.0. This allows the server to execute A_3 to completion at time 19.

Scheduling Aperiodic Jobs with Unknown Execution Times. In the description of the constant utilization server algorithm, we assume that the execution times of aperiodic jobs become known upon their arrival. This restrictive assumption can be removed by modifying the replenishment rules of constant utilization (or total bandwidth) servers. One way is to give the server a fixed size budget e_s and fixed period $e_s/\tilde{u}_s$ just like sporadic and deferrable servers.

Specifically, when an aperiodic job with execution time e shorter than e_s completes, we reduce the current deadline of the server by $(e_s - e)/\tilde{u}_s$ units before replenishing the next e_s units of budget and setting the deadline accordingly. This action clearly can improve the performance of the server and does not make the instantaneous utilization of the server larger than $\tilde{u}_s$.

An aperiodic job with execution time larger than e_s is executed in more than one server period. We can treat the last chunk of such a job in the manner described above if the execution time of this chunk is less than e_s .

7.4.3 Total Bandwidth Server Algorithm

To motivate the total bandwidth server algorithm, let us return to the example in Figure 7–13. Suppose that A_3 were to arrive at time 14 instead. Since 14 is before the current server deadline 15, the scheduler must wait until time 15 to replenish the budget of the constant utilization server. A_3 waits in the interval from 14 to 15, while the processor idles! Clearly, one way to improve the responsiveness of the server is to replenish its budget at time 14. This is exactly what the total bandwidth server algorithm does.

Specifically, the total bandwidth server algorithm improves the responsiveness of a constant utilization server by allowing the server to claim the background time not used by periodic tasks. This is done by having the scheduler replenish the server budget as soon as the budget is exhausted if the server is backlogged at the time or as soon as the server becomes backlogged.

Replenishment Rules of a Total Bandwidth Server of size $\tilde{u}_s$

R1 Initially, $e_s = 0$ and $d = 0$.
R2 When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue, set d to $\max(d, t) + e/\tilde{u}_s$ and $e_s = e$.
R3 When the server completes the current aperiodic job, the job is removed from its queue.
(a) If the server is backlogged, the server deadline is set to $d + e/\tilde{u}_s$, and $e_s = e$.
(b) If the server is idle, do nothing.

Comparing a total bandwidth server with a constant utilization server, we see that for a given set of aperiodic jobs and server size, both kinds of servers have the same sequence of deadlines, but the budget of a total bandwidth server may be replenished earlier than that of a constant utilization server. As long as a total bandwidth server is backlogged, it is always ready for execution.

In the above example, this means that the server's budget is replenished at 6.9 and, if A_3 were to arrive at 14, at 14, and the deadline of the server is 15 and 23.5, respectively, A_3 would be completed at time 17.5 if it were executed by a total bandwidth server but would be completed at 19 by a constant bandwidth server.

Clearly, a total bandwidth server does not behave like a sporadic task with a constant instantaneous utilization. To see why it works correctly, let us examine how the server affects periodic jobs and other servers when its budget is set to e at a time t before the current server deadline d and its deadline is postponed to the new deadline $d' = d + e/\tilde{u}_s$.

If the job $J_{i,k}$ is ready at t , then the amounts of time consumed by both servers are the same in the interval from t to $d_{i,k}$. If $J_{i,k}$ is not yet released at t , then the time demanded by the total bandwidth server in the interval $(r_{i,k}, d']$ is less than the time demanded by the constant utilization server because the total bandwidth server may have executed before $r_{i,k}$ and has less budget in this interval.

COROLLARY 7.7. When a system of independent, preemptable periodic tasks is scheduled with one or more total bandwidth and constant utilization servers on the EDF basis, every periodic task and every server meets its deadlines if the sum of the total density of periodic tasks and the total size of all servers is no greater than 1.

COROLLARY 7.8. When a system of periodic tasks is scheduled with one or more total bandwidth and constant utilization servers on the EDF basis, every periodic task and every server meets its deadlines if the sum of the total density of the periodic tasks and the total size of all servers is no greater than $1 - b_{\max}(np)/D_{\min}$.

COROLLARY 7.9. If the sum of the total density of all the periodic tasks and the total size of total bandwidth and constant utilization servers that are scheduled on the EDF basis is no greater than 1, the tardiness of every periodic task or server is no greater than $b_{\max}(np)$.

7.4.4 Fairness and Starvation

By a scheduling algorithm being fair within a time interval, we mean that the fraction time of processor time in the interval attained by each server that is backlogged throughout the interval is proportional to the server size. For many applications, fairness is important.

To illustrate that this is also true for the total bandwidth server algorithm, let us consider a system consisting solely of two total bandwidth servers, TB_1 and TB_2, each of size 0.5. Each server executes an aperiodic task; jobs in the task are queued in the server's own queue. It is easy to see that if both servers are never idle, during any time interval of length large compared to the execution times of their jobs, the total amount of time each server executes is approximately equal to half the length of the interval. Each server executes for its allocated fraction of time approximately.

Now suppose that in the interval $(0, t)$, for some $t > 0$, server TB_1 remains backlogged, but server TB_2 remains idle. By time t , TB_1 have executed for t units of time and its deadline is at least equal to $2t$. If at time t , a stream of jobs, each with execution time small compared with t , arrives and keeps TB_2 backlogged after t . In the interval $(t, 2t)$, the deadline of TB_2 is earlier than the deadline of TB_1. Hence, TB_2 continues to execute, and TB_1 is starved during this interval.
While processor time is allocated fairly during $(0, 2t)$, the allocation is unfair during $(t, 2t)$. Since t is arbitrary, the duration of unfairness is arbitrary. As a consequence, the response time of jobs executed by TB_1 can arbitrarily be large after t . This is not acceptable for many applications.

Definition of Fairness. Since fairness is not an important issue for periodic tasks, we confine our attention here to systems containing only aperiodic and sporadic jobs. Specifically, we consider a system consisting solely of n (> 1) servers. Each server executes an aperiodic or sporadic task. For $i = 1, 2, \ldots, n$, the size of the ith server is $\tilde{u}_i$.
$\sum_{i=1}^{n} \tilde{u}_i$ is no greater than 1, and hence every server is schedulable.

$\tilde{w}_i(t_1, t_2)$, for $0 < t_1 < t_2$, denotes the total attained processor time of the ith server in the time interval (t_1, t_2), that is, the server executes for $\tilde{w}_i(t_1, t_2)$ units of time during this interval. The ratio $\tilde{w}_i(t_1, t_2)/\tilde{u}_i$ is called the ***normalized service*** attained by the ith server.

A scheduler is *fair* in the interval (t_1, t_2) if the normalized services attained by all servers that are backlogged during the interval differ by no more than the ***fairness threshold*** $FR \geq 0$.

In the ideal case, FR is equal to zero, and

$$\frac{\tilde{w}_i(t_1, t_2)}{\tilde{w}_j(t_1, t_2)} = \frac{\tilde{u}_i}{\tilde{u}_j}$$

for any $t_2 > t_1$ and ith and j th servers that are backlogged throughout the time interval (t_1, t_2).
Equivalently,

$$\tilde{w}_i(t_1, t_2) = \tilde{u}_i(t_2 - t_1)$$

Hence, ideal fairness is not realizable in practice. For a given scheduling algorithm, the difference in the values of the two sides of the above expression depends on the length $t_2 - t_2$ of the time interval over which fairness is measured. In general, FR is a design parameter. By allowing processor time allocation to be somewhat unfair over some time interval length, we admit simple and practical schemes to keep scheduling fair.

Elimination of Starvation. Let us examine again the previous example of two total bandwidth servers. The starvation problem is due to the way in which the total bandwidth server algorithm makes background time available to TB_1; in

general, the deadline of a backlogged total bandwidth server is allowed to be arbitrarily far in the future when there is spare processor time.

A simple solution (to Starvation) is to use only constant utilization servers. ***Since the budget of such a server is never replenished before the current server deadline, the current deadline of a backlogged constant utilization server CU_i of size $\tilde{u}_i$ is never more than $e_{i,max}/\tilde{u}_i$ units of time from the current time.***

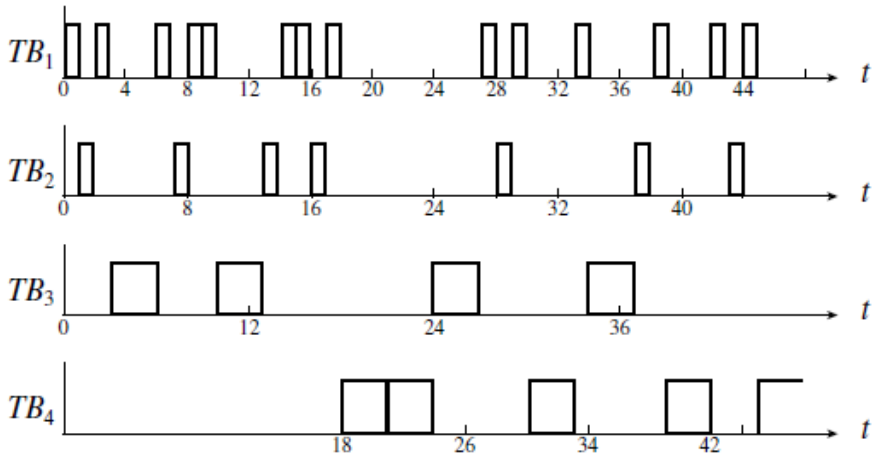
Replenishment Rules of a Starvation-Free Constant Utilization/Background Server

R1 –R3 Within any busy interval of the system, replenish the budget of each backlogged server following rules of a constant utilization server.

R4 Whenever a busy interval of the system ends, replenish the budgets of all backlogged servers.

To illustrate, we look at the example in Figure 7–14. The system contains four aperiodic tasks A_1, A_2, A_3, and A_4, each a stream of jobs with identical execution times.

Specifically, the execution times of jobs in these tasks are 1, 1, 3, and 3, respectively. Each aperiodic task is executed by a server. The sizes of the servers for the tasks are 1/4, 1/8, 1/4, and 3/8, respectively. Starting from time 0, the jobs in A_1, A_2, and A_3 arrive and keep their respective servers backlogged continuously. The first job in A_4 arrives at time 18, and afterwards, the server for A_4 also remains backlogged.



(a) Behavior of total bandwidth servers.

FIGURE 7–14 Example illustrating starvation and fairness: $\tilde{u}_1 = \frac{1}{4}$, $\tilde{u}_2 = \frac{1}{8}$, $\tilde{u}_3 = \frac{1}{4}$, $\tilde{u}_4 = \frac{3}{8}$. $A_1 \equiv$ jobs with execution times $= 1$ arriving from $t = 0$; $A_2 \equiv$ jobs with execution times $= 1$ arriving from $t = 0$; $A_3 \equiv$ jobs with executiong times $= 3$ arriving from $t = 0$; $A_4 \equiv$ jobs with executiong times $= 3$ arriving from $t = 18$.

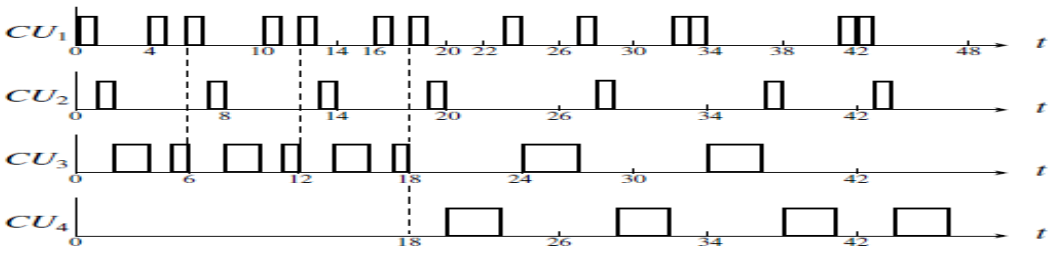
Figure 7–14(a) shows their schedules when the servers are total bandwidth servers. Server TB_i executes A_i, for i = 1, 2, 3, 4. The numbers under each time line labeled by a server name give the deadlines of the server as time progresses. In particular, when the first job in A_4 arrives at time 18, the deadlines of servers TB_1, TB_2, and TB_3 are 36, 40, and 36, respectively. Since the deadline of TB_4 is first set to 26 and, upon the completion of the first job in A_4, to 34, TB_4 executes until 24, starving the other servers in *(*18*,* 24*)*. Before time 18, the amounts of time allocated to the servers according to their sizes are 4.5, 2.25, and 4.5 respectively, but because TB_4 is idle and the time left unused by it is shared by backlogged servers, the servers have executed for 8, 4, and 6 units of time, respectively.

In contrast, their fair shares of processor time should be 7.2, 3.6, and 7.2, respectively. This example supports our intuition: The closer the consecutive deadlines the larger share of background time a server attains at the expense of other backlogged servers.

Now suppose that each aperiodic task A_i is executed by a starvation-free constant utilization/background server CU_i, for i = 1, 2, 3, 4. We have the schedule shown in Figure 7–14(b). At time 6, the budgets of all three backlogged servers are exhausted, and the system becomes idle.

According to rule R4, all three servers gets their new budgets and deadlines. Similarly, their budgets are replenished at time 12. Starting from time 18, all four servers are backlogged, and hence the schedule shown here. It is evident that none of the servers suffers starvation. After time 18, the normalized services of all servers are identical in time intervals of length 24 or more. Before 18, the background time left unused by the idle server is not distributed to backlogged servers in proportion to their sizes, however.

In this example, the servers have executed for 6, 3, and 9 units of time before 18. This illustrates that although the enhanced constant utilization server algorithm eliminates starvation, it does not ensure fairness. It is difficult to determine how the background processor time will be distributed among backlogged server in general.



(b) Behavior of starvation-free constant utilization/background.

FIGURE 7–14 (continued)

7.4.5 Preemptive Weighted Fair-Queueing Algorithm

The well-known **Weighted Fair-Queueing** (WFQ) algorithm is also called the PGPS (packet-by-packet GPS algorithm). It is a nonpreemptive algorithm for scheduling packet transmissions in switched networks. Here, we consider the preemptive version of the weighted fair-queueing algorithm for CPU scheduling and leave the nonpreemptive.

The WFQ algorithm is designed to ensure fairness among multiple servers. The algorithm closely resembles the total bandwidth server algorithm. Both are greedy, that is, work conserving. Both provide the same schedulability guarantee, and hence, the same worst-case response time. The replenishment rules of a WFQ server appear to be the same as those of a total bandwidth server, except for how the deadline is computed at each replenishment time.

This difference, however, leads to a significant difference in their behavior: The total bandwidth server algorithm is unfair, but the WFQ algorithm gives bounded fairness.

Emulation of GPS Algorithm. Again, a WFQ server consumes its budget only when it executes. Its budget is replenished when it first becomes backlogged after being idle. As long as it is backlogged, its budget is replenished each time it completes a job. At each replenishment time, the server budget is set to the execution time of the job at the head of its queue.

In short, the replenishment rules of the WFQ algorithm are such that a WFQ server emulates a GPS server of the same size; the deadline of the WFQ server is the time at which a GPS server would complete the job at the head of the server queue. To illustrate, we look at the example in Figure 7–14 again.

Figure 7–14(c) shows the schedule of the four tasks A_i, for i = 1, 2, 3, 4, when they are scheduled according to the GPS algorithm. Specifically, the figure shows that they are executed by GPS servers GPS_1, GPS_2, GPS_3, and GPS_4, respectively, and the sizes of these servers are 1/4, 1/8, 1/4, and 3/8, respectively. The scheduler schedules backlogged servers on a weighted round-robin basis, with an infinitely small round length and the time per round given to each server is proportional to the server size. The numbers below each time line in Figure 7–14(c) give the completion times of jobs executed by the respective server.