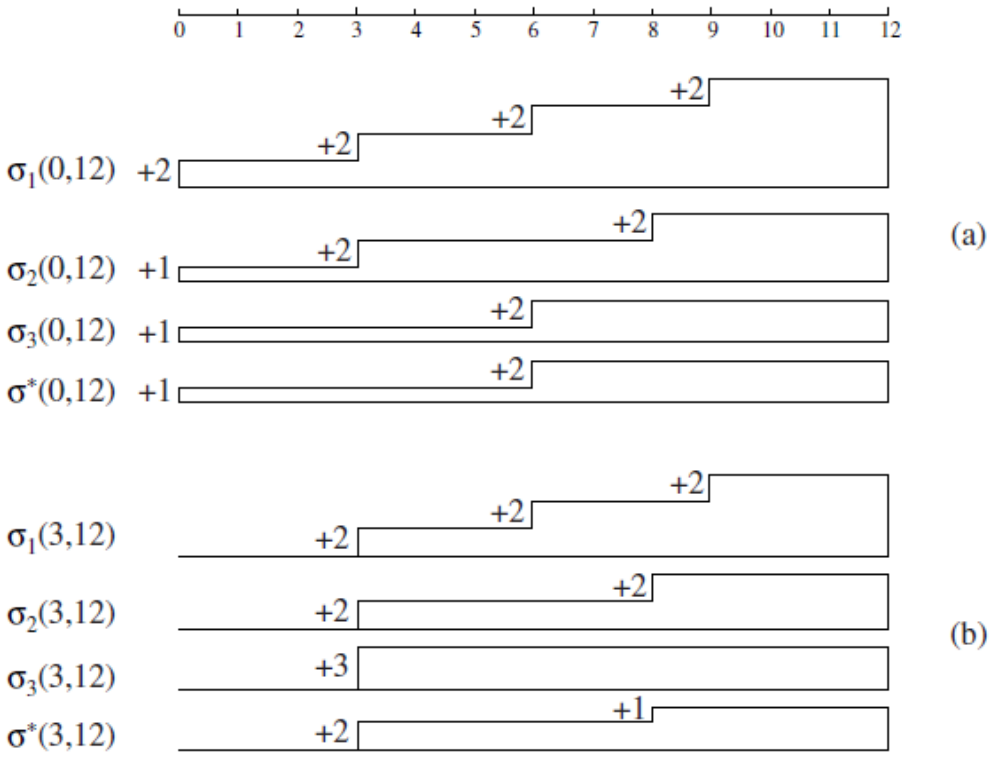

FIGURE 7–20 Slack functions.

Computation Procedure. From Eq. (7.10) and Figure 7–20, we see that in general, the slack function $\sigma^*(t_c, t)$ of each task T_i is a staircase function and it has a step (increase) at the effective deadline of each job in T_i . Consequently, the minimum slack function $\sigma^*(t_c, t)$ is also a staircase function. Let $y_1 < y_2 < \cdots < y_k \ldots$ be the values of t at which $\sigma^*(t_c, t)$ increases. Each y_k is the effective deadline of some periodic job. With a slight abuse of the term, we call these time instants the steps of $\sigma^*(t_c, t)$. A computation of $\sigma^*(t_c, t)$ amounts to finding the steps of the function and the increase in slack at each step.

According to the algorithm proposed by Tia, each slack computation finds the next step of the minimum slack function and the value of the function until the step. The initial step y_0 occurs immediately after the beginning of the current hyperperiod, and the slack of each task T_i at the initial step is equal to $\sigma_{i,1}(0)$.

Let y_{k-1} ($k > 0$) denote the step of the minimum slack function found during the initial or the previous slack computation. z_i denotes the next step of $\sigma_i(t_c, t)$, that is, the earliest step of this function after $\max(t_c, y_{k-1})$. $z_{\min}$ denotes the step among z_i for $1 \le i \le n$ that has the least slack. In other words, $\sigma_{\min}(t_c, z_{min}-) \le \sigma_i(t_c, z-i)$ for all i. If there is a tie, $z_{\min}$ is the latest among the z_i 's that have the minimum slack. The next step y_k is equal to $z_{\min}$. The minimum slack function $\sigma^*(t_c, t)$ is equal to $\sigma_{\min}(t_c, z_{min}-)$ and is the slack of the system until y_k . The job in $T_{\min}$ whose effective deadline is $z_{\min}$ is called the ***constraining job*.**

As an example, suppose that the scheduler does a slack computation at time 1. The next step z_i of $\sigma_i(1, t)$ is equal to 3, 3, and 6 for i = 1, 2, and 3, respectively. Immediately before their corresponding next steps, the slack functions of the tasks are equal to 2, 2, and 1, respectively. Hence, the next step y_1 of the minimum slack function is $z_3 = 6$. $\sigma^*(t_c, y_1)$ is 1. The constraining job is $J_{3,1}$.

7.6.3 Scheduling Aperiodic Jobs

In addition to maintaining I , ST and ξ_i for $i = 1, 2, \ldots, n$ as stated above, the scheduler also updates the remaining execution time e_A of the aperiodic job A at the head of the aperiodic job queue if the job is preempted before it is

completed. The scheduler controls when any available slack is used by varying the priority of the slack stealer relative to the fixed priorities of the periodic tasks. We have the following two observations about the constraining job J found at the time t_c of the kth slack computation.

1. There can be no more than $\sigma^*(t_c, y_k)$ units of slack before y_k unless the constraining job J is completed before y_k .

2. Scheduling the constraining job J so that it completes as soon as possible will not lead to a decrease in the amount of slack before y_k .

If J completes at some time t before y_k , we may be able to delay the execution of some periodic jobs in $\mathbf{H}_i(t_c, y_k)$ until after y_k and thus create more slack before y_k .

Scheduler Operations. These observations are the rationales behind the slack stealing algorithm defined by the following rules.

R0 Maintain history information, that is, I , SI, and ξ_i, for $i = 1, 2, \ldots, n$, and update e_A before A starts execution and when it is preempted.

R1 Slack Computation: Carry out a slack computation each time when

- an aperiodic job arrives if the aperiodic job queue is empty prior to the arrival,
- an aperiodic job completes and the aperiodic job queue remains nonempty, and
- a constraining job completes.

The next step and minimum slack found by the slack computation are y, y being the effective deadline of a job in $T_{\min}$, the available slack until y is $\sigma^*(t_c, y) = \sigma_{\min}(t_c, y)$.

R2 Assigning Priority to Slack Stealer: Each time following a slack computation, assign the slack stealer the highest priority if $\sigma^*(t_c, y) \geq e_A$. Otherwise, the priority of the slack stealer is between that of $T_{\min}$ and $T_{\min+1}$.

R2 Periodic jobs and slack stealers execute according to their priorities.

We again use the example in Figure 7–19 to illustrate. Suppose that shortly after time 0, an aperiodic job A with execution time equal to 1.5 arrives. The slack functions of all tasks being as shown in Figure 7–20(a), the next step y is 6, the effective deadline of constraining job $J_{3,1}$. Since the $\sigma^*(0, 6)$ (= 1) is less than the remaining execution time e_A (= 1.5), the slack stealer is given a lower priority than $J_{3,1}$. Upon the completion of $J_{3,1}$ at 3, the slack is computed again.

This time the slack of the system is equal to 2. Therefore, the slack stealer is given the highest priority. It starts to execute at 3 and completes A at 4.5. On the other hand, suppose that the execution time of A is 2.5. At time 3, the slack stealer is given a priority lower than the constraining job $J_{2,2}$ but higher than $J_{3,2}$. This allows $J_{2,2}$ to complete at 5. When the slack is completed again at 5, the system has 3 units of slack. Consequently, the slack stealer is given the highest priority and it completes A at time 7.5.

Performance. The slack-stealing approach gives smaller response times when compared with the sporadic server scheme even when the sporadic server is allowed to use background time. This is especially true when the total utilization of periodic tasks is higher than 75 percent. When the total utilization of periodic tasks is this large, one is forced to give a sporadic server a small size. So, the performance of a sporadic server becomes closer to that of a background server. But, by letting the slack stealer execute aperiodic jobs while the periodic tasks have slack, one can reduce their response times significantly.

7.7 SCHEDULING OF SPORADIC JOBS

We assume that acceptance tests are performed on sporadic jobs in the EDF order. Once accepted, sporadic jobs are ordered among themselves in the EDF order. In a deadline-driven system, they are scheduled with periodic jobs on the EDF basis. In a fixed-priority system, they are executed by a bandwidth preserving server. In both cases, no new scheduling algorithm is needed. In our subsequent discussion, we refer to each individual sporadic job as S_i.

When we want to call attention to the fact that the job is released at time t and has maximum execution time e and (absolute) deadline d, we call the job S_i *(t, d, e)*. We say that an acceptance test is optimal if it accepts a sporadic job if and only if the sporadic job can be feasibly scheduled without causing periodic jobs or sporadic jobs in the system to miss their deadlines.

7.7.1 A Simple Acceptance Test in Deadline-Driven Systems

Theorem 7.4 says that in a deadline-driven system where the total density of all the periodic tasks is Δ, all the accepted sporadic jobs can meet their deadlines as long as the total density of all the active sporadic jobs is no greater than 1−Δ at all times. This fact gives us a theoretical basis of a very simple acceptance test in a system where both the periodic and sporadic jobs are scheduled on the EDF basis.

Acceptance Test Procedure. The acceptance test on the first sporadic job *S(t, d, e)* is simple indeed. The scheduler accepts S if its density $e/(d - t)$ is no greater than $1 - \Delta$. If the scheduler accepts the job, the (absolute) deadline d of S divides the time after t into two disjoint time intervals: the interval I_1 at and before d and the interval I_2 after d. The job S is active in the former but not in the latter. Consequently, the total densities $\Delta_{s,1}$ and $\Delta_{s,2}$ of the active sporadic jobs in these two intervals are equal to $e/(d - t)$ and 0, respectively.

We now consider the general case. At time t when the scheduler does an acceptance test on *S(t, d, e)*, there are n_s active sporadic jobs in the system. For the purpose of scheduling them and supporting the acceptance test, the scheduler maintains a non-decreasing list of (absolute) deadlines of these sporadic jobs. These deadlines partition the time interval from t to the infinity into $n_s + 1$ disjoint intervals: $I_1, I_2, \ldots, I_{ns+1}$. I_1 begins at t and ends at the first (i.e., the earliest) deadline in the list. For $1 \le k \le ns$, each subsequent interval I_{k+1} begins when the previous interval I_k ends and ends at the next deadline in the list or, in the case of I_{ns+1}, at infinity.

The scheduler also keeps up-to-date the total density $\Delta_{s,k}$ of the sporadic jobs that are active during each of these intervals.

Let I_l be the time interval containing the deadline d of the new sporadic job *S(t, d, e)*. Based on Theorem 7.4, the scheduler accepts the job S if

$$\frac{e}{d-t} + \Delta_{s,k} \le 1 - \Delta \qquad (7.11)$$

for all $k = 1, 2, \ldots l$.
If these conditions are satisfied and S is accepted, the scheduler divides the interval I_l into two intervals:
The first half of I_l ends at d, and the second half of I_l begins immediately after d. We now call the second half I_l+1 and rename the subsequent intervals $I_{l+2}, \ldots, I_{ns+2}$.

The scheduler increments the total density $\Delta_{s,k}$ of all active sporadic jobs in each of the intervals $I_1, I_2, \ldots, I_l$ by the density $e/(d - t)$ of the new job. Thus, the scheduler becomes ready again to carry out another acceptance test. The complexity of this acceptance test is $O(N_s)$ where N_s is the maximum number of sporadic jobs that can possibly be in the system at the same time.

As an example, we consider a deadline-driven system in Figure 7–21. The system has two periodic tasks $T_1 = (4, 1)$ and $T_2 = (6, 1.5)$. The relative deadline of each task is equal to the period of the task. Their total density is 0.5, leaving a total density of 0.5 for sporadic jobs.

1. Suppose that the first sporadic job $S_1(0+, 8, 2)$ is released shortly after time 0. Since the density of S_1 is only 0.25, the scheduler accepts it. The deadline 8 of S_1 divides the future time into two intervals: $I_1 = (0+, 8]$ and $I_2 = (8,\infty)$. The scheduler updates the total densities of active sporadic jobs in these intervals: $\Delta_{s,1} = 0.25$ and $\Delta_{s,2} = 0$. The scheduler then inserts S_1 into the queue of ready periodic and sporadic jobs in the EDF order.

2. At time 2, the second sporadic job $S_2(2, 7, 0.5)$ with density 0.1 is released. Its deadline 7 is in I_1. Since the condition $0.1 + 0.25 \le 0.5$ defined by Eq. (7.11) is satisfied, the scheduler accepts and schedules S_2. We now call the interval I_2 I_3. The interval I_1 is divided into $I_1 = (2, 7]$ and $I_2 = (7, 8]$. The scheduler increments the total density $\Delta_{s,1}$ by the density of S_2. Now, $\Delta_{s,1} = 0.35$, $\Delta_{s,2} = 0.25$, and $\Delta_{s,3} = 0$.

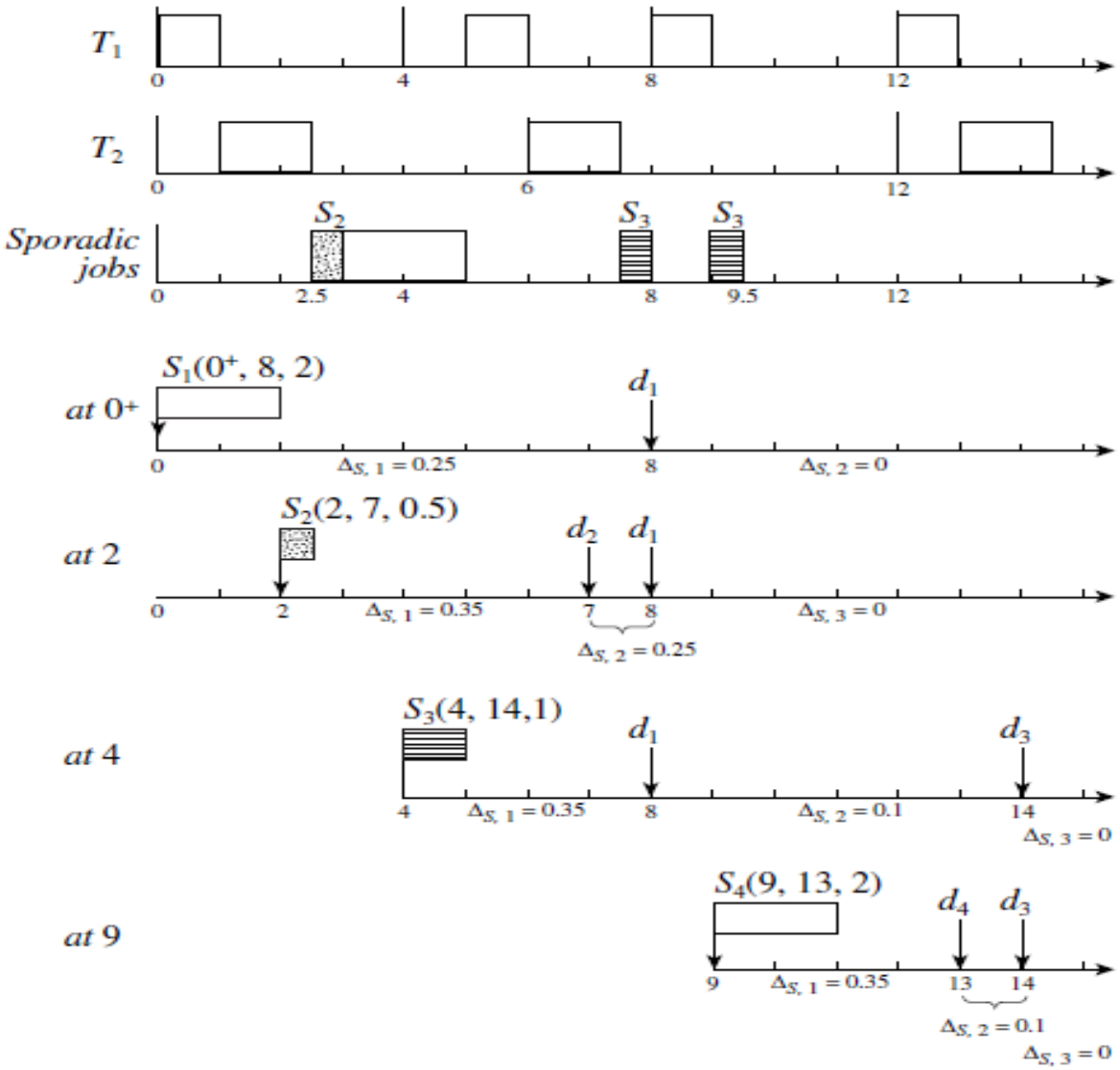

FIGURE 7–21 An acceptance test on $T_1 = (4, 1)$, and $T_2 = (6, 1.5)$.

3. At time 4, $S_3(4, 14, 1)$ is released. S_2 has already completed. The only sporadic job in the system is S_1. $\Delta_{s,1} = 0.25$ and $\Delta_{s,2} = 0$. The deadline of S_3 is in the interval I_2. The conditions the scheduler checks are whether 0.25 + 0.1 and 0.1 are

equal to or less than 0.5. Since both are satisfied, the scheduler accepts S_3. The intervals maintained by the scheduler are now $I_1 = (4, 8]$, $I_2 = (8, 14]$, and $I_3 = (14, \infty]$. So, $\Delta_{s,i}$ are 0.35, 0.1, and 0 for $i = 1, 2$, and 3, respectively.

4. At time 9, $S_4(9, 13, 2)$ is released. Now, the only active sporadic job in the system is S_3. I_1 is $(9, 14]$ and I_2 is $(14, \infty)$. Since for I_1, the total density of existing active sporadic jobs is 0.1 and the density of S_4 is 0.5, their sum exceeds 0.5. Consequently, the scheduler rejects S_4.

Enhancements and Generalization. Two points are worthy of observing. ***First***, the acceptance test is not optimal, meaning that a sporadic job may be rejected while it is schedulable. An enhancement is to have the scheduler also compute the slack of the system. However, that acceptance test can be used only when periodic tasks have little or no release-time jitters.

The ***second*** point worth noting is that the acceptance test described above assumes that every sporadic job is ready for execution upon its arrival. The ready times and deadlines of sporadic jobs in the system partition the future time into disjoint time intervals. The scheduler maintains information on these intervals and the total densities of active sporadic jobs in them in a similar manner, but now it may have to maintain as many as $2N_s + 1$ intervals, where N_s is the maximum number of sporadic jobs in the system.

***7.7.2 An Acceptance Test Based on Slack Computation in Deadline-Driven Systems**

We now describe an algorithm that uses static-slack computation and is for systems where periodic and accepted sporadic jobs are scheduled on the EDF basis. The time complexity of the acceptance test based on this algorithm is $O(n + Ns)$.

Major Steps. We call the ith periodic job in each hyperperiod J_i for $i = 1, 2, \ldots, N$. The N jobs are indexed so that the deadline d_i of J_i is equal to or less than the deadline d_k of J_k if $i < k$. Suppose that at the time t when an acceptance test is done on a new sporadic job S, there are also n_s sporadic jobs in the system.

We call the sporadic jobs $S_1, S_2, \ldots, S_{ns}$. Let $e_{s,k}$ and $d_{s,k}$ denote the maximum execution time and absolute deadline of S_k , respectively. We again assume that every sporadic job is ready for execution as soon as it is released. We continue to use $\sigma_k(t)$ to denote the slack of the periodic job J_k at time t in the absence of any sporadic jobs. The notation for the slack of the sporadic job S_k at t is $\sigma_{s,k}(t)$. When a sporadic job $S_i(t, d, e)$ is released at time t , the scheduler carries out the following steps to accept and schedule S_i or to reject S_i .

1. The scheduler computes the slack $\sigma_{s,i}(t)$ of the tested job S_i. If $\sigma_{s,i}(t)$ is less than 0, it rejects S_i . Otherwise, it carries out the next step.
2. The scheduler accepts S_i and carries out step 3 if the current slack of every job whose deadline is at or after d is at least equal to the execution time e of S_i ; otherwise, it rejects S_i .
3. When the scheduler accepts $S_i(t, e, d)$, it inserts the job into the EDF priority queue of all ready jobs, allocates a register for the storage of the execution time $\xi_{s,i}$ of the completed portion of S_i , stores the initial slack $\sigma_{s,i}(t)$ of S_i , and decrements the slack of every existing sporadic job whose deadline is at or after d by e.

To support slack computation, the scheduler updates the total amount TE of time in the current hyperperiod. It also maintains the total amount I of time the system has been been idle since the beginning of the current hyperperiod, as well as the execution time ξ_k (or $\xi_{s,k}$) of the completed portion of each current periodic (or existing sporadic) job in the system.

(Finally, in this description, by a sum over sporadic jobs, we mean sporadic jobs that are still in the system; the completed sporadic jobs are not included.)

For a given sporadic job $S_i(t, d, e)$ being tested, we let J_l denote the periodic job whose deadline d_l is the latest among

all the periodic jobs whose deadlines are at or before d, if there are such jobs. This job is called the ***leverage job*** of S_i . If the deadline d of S_i is earlier than the deadline of the first periodic job in the hyperperiod, its leverage job is a nonexisting periodic job.

Acceptance Test for the First Sporadic Job. To compute the slack of the first sporadic job $S_1(t, d, e)$ tested in the current hyperperiod, the scheduler first finds its leverage job J_l . In addition to the slack of J_l , which can be used to execute S_1, the interval $(d_l , d]$ is also available. Hence the slack of S_1 is given by

$$\sigma_{s,1}(t) = \sigma_l(t) + (d - d_l) - e \qquad (7.12a)$$

The slack of the job J_l in the absence of any sporadic job is given by Eq. (7.7) if there are aperiodic jobs in the system and a slack stealer to execute them. Otherwise, if there is no slack stealer, as we have assumed here,

$$\sigma_l(t) = \omega(l; l) - I - \sum_{d_k > d} \xi_k \qquad (7.12b)$$

When the deadline of the first periodic job in the hyperperiod is later than d, $\sigma_l(t)$ and d_l are zero by definition, and $\sigma_{s,1}(t)$ is simply equal to $d - e$.

In the first step of the acceptance test, the scheduler computes $\sigma_{s,1}(t)$ in this way. It rejects the new sporadic job S_1 if $\sigma_{s,1}(t)$ is less than 0. Otherwise, it proceeds to check whether the acceptance of the S_1 may cause periodic jobs with deadlines after d to miss their deadlines.

For this purpose, it computes the minimum of the current slacks $\sigma_k(t)$ of all the periodic jobs J_k for $k = l + 1, l + 2, \ldots, N$ whose deadlines are after d. The minimum slack of these jobs can be found using the static-slack computation method This computation and the subsequent actions of the scheduler during the acceptance test also take $O(n)$ time.

Acceptance Test for the Subsequent Sporadic Jobs. In general, when the acceptance of the sporadic job S_i (t, d, e) is tested, there may be n_s sporadic jobs in the system waiting to complete. Similar to Eq. (7.12), the slack of S_i at time t can be expressed in terms of the slack $\sigma_l(t)$ of its leverage job J_l as follows.

$$\sigma_{s,l}(t) = \sigma_l(t) + (d - d_l) - e - TE - \sum_{d_{s,k} \le d} e_{s,k} - \sum_{d_{s,k} > d} \xi_{s,k} \qquad (7.13)$$

After computing the slack $\sigma_{s,i}(t)$ available to S_i in this manner, the scheduler proceeds to the second step of the acceptance test if $\sigma_{s,i}(t) \ge 0$.

This time, the second step has two substeps. First, the scheduler checks whether the acceptance of S_i would cause any yet-to-be completed sporadic job S_k whose deadline is after d to be late. This can be done by comparing the current slack $\sigma_{s,k}(t)$ of S_k with the execution time e of the new job S_i . S_i is acceptable only when the slack of every affected sporadic job in the system is no less than e.
If the new job S_i (t, d, e) passes the first substep, the scheduler then checks whether the minimum of current slacks of periodic jobs whose deadlines are at or after d is at least as large as e. Specifically, the periodic jobs are now partitioned into at most $n+n_s$ disjoint subsets.

Let $\mathbf{Z}_j$ denote one of these subsets and J_x and J_y denote the periodic jobs that have the earliest deadline d_x and latest deadline d_y , respectively, among the jobs in this subset. The minimum slack of all periodic jobs in $\mathbf{Z}_j$ is given by

$$\omega_j(t) = \omega(x; y) - I - TE - \sum_{d_{s,k} \le d_x} e_{s,k} - \left(\sum_{d_{s,k} > d_y} \xi_{s,k} + \sum_{d_k > d_y} \xi_k \right) \qquad (7.14)$$

Clearly, the minimum slack of all periodic jobs whose deadlines are after *d* is

$$\min_{1 \le j \le n+n_s} \omega_j(t).$$

The scheduler accepts the new job S_i *(t, d, e)* only if this minimum slack is no less than *e*. The slack computation takes $O(n + n_s)$ time.

Acceptance Tests for Sporadic Jobs with Arbitrary Deadlines. Finally, let us consider a sporadic job S_i *(t, d, e)* that arrives in the current hyperperiod and its deadline *d* is in some later hyperperiod. Without loss of generality, suppose that the current hyperperiod starts at time 0 and the deadline *d* is in *z*th hyperperiod.

In other words, $0 < t \le (z - 1)H < d \le zH$. For this sporadic job, the leverage job J_l is the last job among all the periodic jobs that are in the *z*th hyperperiod and have deadlines at or before *d*, if there are such jobs, or a nonexisting job whose slack is 0 and deadline is $(z - 1)H$, if there is no such periodic job.

In this case, the time available to the new job S_i consists of three terms: the slack a_1 in the current hyperperiod, the slack a_2 in the second through $(z - 1)$th hyperperiods and the slack a_3 in the *z*th hyperperiod. a_1 is given by

$$a_1 = \sigma_N(0) - I - TE - \sum_{d_{s,k} \le d_N} e_{s,k} - \sum_{d_{s,k} > d_N} \xi_{s,k} \qquad (7.15a)$$

Similarly,

$$a_2 = (z - 2)\sigma_N(0) - \sum_{d_N < d_{s,k} \le (z-1)d_N} e_{s,k} \qquad (7.15b)$$

The time in the *z*th hyperperiod available to S_i is equal to

$$a_3 = \sigma_l(0) + (d - d_l) - \sum_{(z-1)H < d_{s,k} \le d_l} e_{s,k} \qquad (7.15c)$$

In the first step, the scheduler computes the available slack of the new job $S_i(t, d, e)$ according to

$$\sigma_{s,i}(t) = a_1 + a_2 + a_3 - e \qquad (7.15d)$$

The scheduler proceeds to do step 2 of the acceptance test if $\sigma_{s,i}$ *(t)* is no less than 0. In the ***second step***, the scheduler only needs to check whether the periodic jobs that are in the *z*th hyperperiod and have deadlines at or after *d* may be adversely affected by the acceptance of S_i .

The periodic tasks in the system shown in Figure 7-21 are $T_1 = (4, 1)$ and $T_2 = (6, 1.5)$. The length *H* of each hyperperiod is 12, and *N* is equal to 5. The initial slack σ_i *(0)* of the five periodic jobs are 3.0, 3.5, 4.5, 7.0, and 6.0, respectively, for *i* equal to 1, 2, 3, 4, and 5.

1. At time 0^+, $S_1(0^+, 8, 2)$ is tested. For this sporadic job, the leverage job is J_2 (i.e., $J_{2,1}$ with deadline 6). In the first step, the scheduler finds $\sigma_{s,1}(0^+)$, which is equal to $\sigma_2(0) + (8 - 6) - 2 = 3.5$ according to Eq. (7.12a). In the second step, it finds the minimum slack of periodic jobs J_3, J_4, and J_5. This minimum is equal to 2.5 if S_1 is accepted.

Hence, the scheduler accepts S_1, inserts it into the EDF queue of ready periodic and sporadic jobs, stores $\sigma_{s,1}(0^+) = 3.5$ for later use. It also creates a register to store the current value of $\xi_{s,1}$ of S_1.

2. At time 2, $S_2(2, 7, 0.5)$ is tested. Its leverage job J_l is also J_2. Up until time 2, the system has never been idle and no sporadic job has completed. Hence, I and TE are both 0. $\xi_{s,1}$ is also equal to 0. No periodic job with deadline after 7 has executed.

According to Eq. (7.13), $\sigma_{s,2}(2)$ is equal to 3.5 + 1 − 0.5 = 4.0. Since it is larger than 0.5, the scheduler proceeds to step 2 of the acceptance test. Since none of the jobs with deadlines after 7 will be adversely affected, the scheduler accepts $S2$. In addition to do all the bookkeeping work on $S2$, the scheduler updates the slack of S_1: $\sigma_{s,1}(2) = 3.5 - 0.5 = 3.0$.

3. By time 4.0 when $S_3(4, 14, 1)$ is released and tested, S_2 has completed. TE is therefore equal to 0.5. I is still zero. The deadline of S_3 is in the second hyperperiod. The scheduler first computes a_1 according to Eq. (7.15a): $a1 = 6 - 0.5 - 2 = 3.5$. a_2 is equal to zero. Since no existing sporadic job has a deadline in the second hyperperiod, according to Eq. (7.15c), $a3$ is equal to 2. Hence $\sigma_{s,3}(4) = 3.5 + 2 - 1 = 4.5$.

In the second step, the scheduler needs to check only whether the jobs with deadlines in the second hyperperiod would be late if $S3$ is accepted. Since all of their slacks are larger than the execution time of $S3$, $S3$ is acceptable and is accepted.

4. When $S_4(9, 13, 2)$ is tested at time 9, $I = 0$, and $TE = 2.5$. There is only one sporadic job S_3 in the system, and $\xi_{s,3}(9) = 0.5$. The scheduler finds $a1$ equal to $6 - 0 - 2.5 - 0.5 = 3.0$. a_3 is equal to 1.0. Hence the slack $\sigma_{s,4}(9) = 3 + 1 - 2 = 2$. Since the slack $\sigma_{s,3}(9)$ is 4.5, S_3 will not be late if S_4 is accepted.

We observe that if the execution time of $S4$ were 4.0, this test would accept the job, and the action would be correct.

7.7.3 A Simple Acceptance Test in Fixed-Priority Systems

One way to schedule sporadic jobs in a fixed-priority system is to use a sporadic server to execute them. Because the server (p_s, e_s) has e_s units of processor time every p_s units of time, the scheduler can compute the least amount of time available to every sporadic job in the system. This leads to a simple acceptance test.

We assume that accepted sporadic jobs are ordered among themselves on an EDF basis. When the first sporadic job $S_1(t, d_{s,1}, e_{s,1})$ arrives, the server has at least $\lfloor (d_{s,1} - t)/p_s \rfloor e_s$ units of processor time before the deadline of the job. Therefore, the scheduler accepts S_1 if the slack of the job

$$\sigma_{s,1}(t) = \lfloor (d_{s,1} - t)/p_s \rfloor e_s - e_{s,1}$$

is larger than or equal to 0.

To decide whether a new job $S_i(t, d_{s,i}, e_{s,i})$ is acceptable when there are n_s sporadic jobs in the system, the scheduler computes the slack $\sigma_{s,i}$ of S_i according to

$$\sigma_{s,i}(t) = \lfloor (d_{s,i} - t)/p_s \rfloor e_s - e_{s,i} - \sum_{d_{s,k} < d_{s,i}} (e_{s,k} - \xi_{s,k})$$

where $\xi_{s,k}$ is the execution time of the completed portion of the existing sporadic job S_k. The new job S_i cannot be accepted if its slack is less than 0.

If $\sigma_{s,i}(t)$ is no less than 0, the scheduler then checks whether any existing sporadic job S_k whose deadline is after $d_{s,i}$ may be adversely affected by the acceptance of S_i. The scheduler accepts S_i if $\sigma_{s,k}(t) - e_{s,i} \geq 0$ for every existing sporadic job S_k with deadline equal to or later than $d_{s,i}$. If the scheduler accepts S_i, it stores $\sigma_{s,i}(t)$ for later use and decrements the slack of every sporadic job with a deadline equal to or later than $d_{s,i}$ by the execution time $e_{s,i}$ of the new job.

7.7.4 Integrated Scheduling of Periodic, Sporadic, and Aperiodic Tasks

In principle, we can schedule sporadic and aperiodic tasks together with periodic tasks according to either the bandwidth-preserving server approach or the slack-stealing approach, or both. However, it is considerably more complex to do slack stealing in a system that contains both sporadic tasks with hard deadlines and aperiodic tasks.

If we steal slack from sporadic and periodic jobs in order to speed up the completion of aperiodic jobs, some sporadic jobs may not be acceptable later. The presence of sporadic jobs further increases the complexity of slack computation.

7.8 REAL-TIME PERFORMANCE FOR JOBS WITH SOFT TIMING CONSTRAINTS

For many applications, occasional missed deadlines are acceptable; their sporadic jobs have soft deadlines.

7.8.1 Traffic Models

In a system that provides each sporadic task with some kind of performance guarantee, the system subjects each new sporadic task to an acceptance test. Once a sporadic task is admitted into the system, the scheduler accepts and schedules every job in it. When requesting admission into the system, each sporadic task presents to the scheduler its ***traffic parameters***.

These parameters define the constraints on the interarrival times and execution times of jobs in the task. The performance guarantee provided by the system to each task is ***conditional***, meaning that the system delivers the guaranteed performance conditioned on the fact that the task meets the constraints defined by its traffic parameters.

The flip side is that if the task misbehaves, the performance experienced by it may deteriorate. In addition the periodic task and sporadic task models, there are also the Ferrari and Verma (FeVe) model [FeVe], the *(λ, ϐ)* model [Cruz], and the leaky bucket model [Turn, ClSZ]. These models are commonly used to characterize real-time traffic in communication networks.

FeVe and (*λ, ϐ*) Models. According to the FeVe model, each sporadic task is characterized by a 4-tuple

$$(e, p, \overline{p}, I):$$

e is the maximum execution time of all jobs in the task; *p* is the minimum interarrival time of the jobs; $\overline{p}$ is their average interarrival time, and this average is taken over a time interval of length *I*.

Each job may model the transmission of a message (or packet) and each sporadic task the transmission of a message stream. It is also a good model of other types of sporadic tasks (e.g., computations) when the execution times of jobs in each task are roughly the same but their interarrival times may vary widely. The *(λ, ϐ)* model, characterizes each sporadic task by a rate parameter *λ* and a burst parameter *ϐ*. The total execution time of all jobs that are released in any time interval of length *x* is no more than $ϐ + λx$.

Leaky BucketModel. To define the *leaky bucket model*, we first introduce the notion of a *(Λ, E)* leaky bucket filter. Such a filter is specified by its (input) rate *Λ* and size *E*: The filter can hold at most *E* tokens at any time and it is being filled with tokens at a constant rate of *Λ* tokens per unit time. A token is lost if it arrives at the filter when the filter already contains *E* tokens.

We can think of a sporadic task that meets the *(Λ, E)* leaky bucket constraint as if its jobs were generated by the filter in the following manner. The filter may release a job with execution time *e* when it has at least *e* tokens. Upon the release of this job, *e* tokens are removed from the filter. No job can be released when the filter has no token. Therefore, no job in a sporadic task that satisfies the *(Λ, E)* leaky bucket constraint has execution time larger than *E*.

The total execution time of all jobs that are released within any time interval of length E/Λ is surely less than $2E$. A periodic task with period equal to or larger than E/Λ and execution time equal to or less than E satisfies the (Λ, E) leaky bucket constraint. A sporadic task $S = (1, p, \bar{p}, I)$ that fits the FeVe model satisfies this constraint if $\bar{p} = 1/\Lambda$ and $E = (1 - p/\bar{p})\, I/\bar{p}$.

Figure 7–22 shows an example. The arrows above the time line indicate the release times of jobs of a sporadic task in a time interval of length 70. The execution times of the jobs are given by the numbers above the boxes symbolizing the jobs, and their relative deadlines are all equal to 15.

• Of course, we can model the task as a periodic task $(5, 2, 15)$ since the minimum interrelease time is 5 and the maximum execution time of all instances is 2. However, this

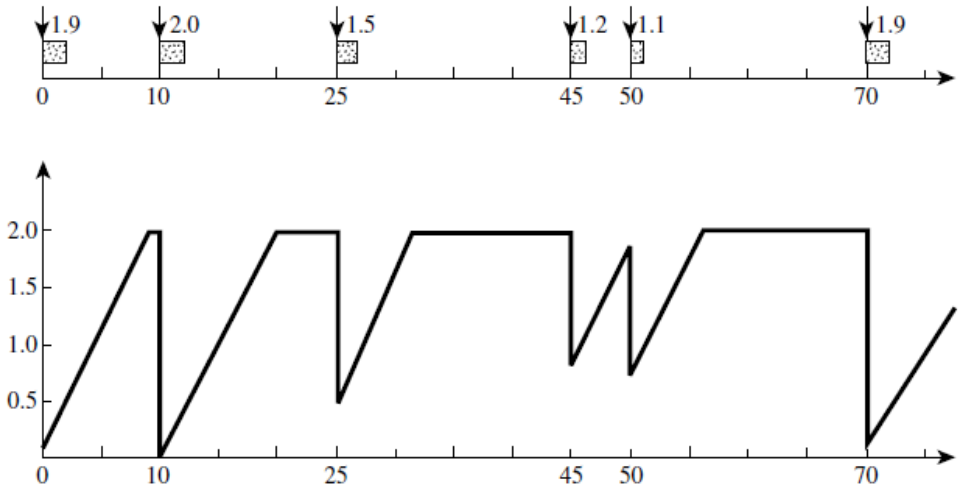

FIGURE 7–22 Example illustrating the leaky bucket traffic model.

is not an accurate characterization; the utilization of this periodic task is 0.4, which is many times larger than its average utilization of 0.11.

• Following the sporadic task model, we can characterize the task by the sequence of instantaneous utilizations of its jobs: 0.19, 0.133, 0.075, 0.24, 0.055, and so on. The maximum instantaneous utilization of the stream is 0.24.

• This task satisfies the $(0.2, 2.0)$-leaky bucket constraint. The diagram at the bottom of the figure shows the number of token remaining in a $(0.2, 2.0)$-leaky bucket filter as a function of time. The bucket is full at time 0. In fact, the task satisfies this constraint provided the size of the leaky bucket is no less than 2.0 and its rate is no less than 0.19.

7.8.2 Performance of Bandwidth-Preserving Server Algorithms

Because no job of any accepted sporadic task is rejected by the scheduler, it is possible for a sporadic task to overload the processor. So, the scheduler must provide firewall protection. The bandwidth-preserving server algorithms are designed to provide such protection and, therefore, are ideally suited for scheduling sporadic tasks with soft deadlines.

We focus here on systems which use a bandwidth-preserving server to execute each sporadic task. To know the maximum response time of jobs in a sporadic task that is executed by a constant utilization, total bandwidth, or WFQ server in a deadline-driven system. Clearly, the size $\tilde{u}$ of the server must at least be equal to the average utilization u of the sporadic task S.

Queueing theory tells us that even when $\tilde{u} = u$, the average response time of the jobs in S is unbounded if their execution times and interarrival times are not constrained.

$(e, p, \bar{p}, I)$-Constrained Sporadic Tasks. Suppose that the sporadic task fits the FeVe model: $S = (e, p, \bar{p}, I)$, and $u = e/\bar{p}$. We assume that the number of jobs in S released in any interval of length I is never more than $I/\bar{p}$.

Corollary 7.7 tells us as long as the total server size is no greater than 1, we can answer this question without regard to jobs executed by other servers. To find the maximum possible response time W of all jobs in S, we look at a busy interval of the server that begins at the arrival time t_0 of the first of a bursty sequence of X jobs. By their arrivals being bursty, we mean that the interarrival times of these X jobs are all

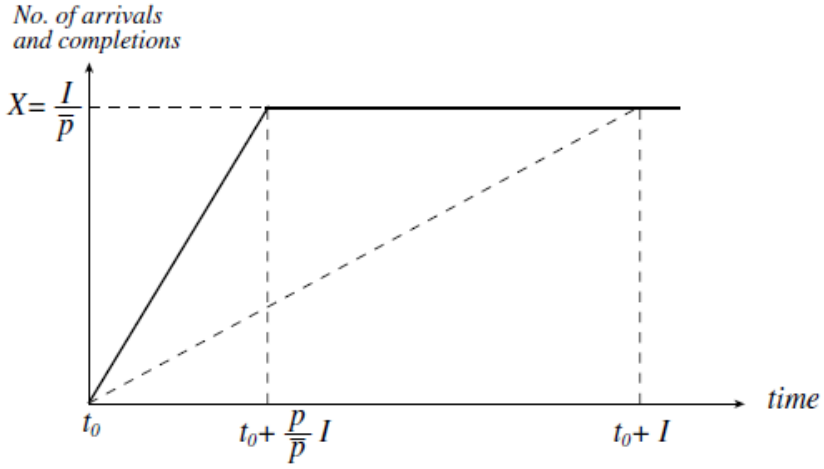

FIGURE 7–23 Number of arrivals and completions.

equal to the minimum possible value p. Because S meets the $(e, p, \bar{p}, I)$ constraint, $X \le I/p$.

The solid line in Figure 7–23 shows the number of arrivals of jobs in S as a function of time from t_0, and the dotted line in the figure shows the number of completions of these jobs. The vertical difference between the two lines at any time gives the number of jobs waiting to be executed at the time. It is evident that the response time of the Xth job in the busy interval is larger than the response times of all other jobs executed in the busy interval.

To find the response time of the Xth job we note that immediately before the release time of the Xth job in S, there are no more than $(X-1) - \lfloor (X-1)p/\bar{p} \rfloor$ jobs waiting to be completed. Since it may take the server p units of time to complete each job, maximum response time of the Xth job is bounded from above by

$$W = \begin{cases} \left(X - \dfrac{(X-1)p}{\bar{p}}\right)\bar{p} + \bar{p} = p + \bar{p} + I\left(1 - \dfrac{p}{\bar{p}}\right) & \text{if } p < \bar{p} \\ \bar{p} & \text{if } p = \bar{p} \end{cases} \qquad (7.16)$$

$(1/\bar{p}, I/\bar{p})$-Leaky Bucket Constrained Tasks. An important special case is when the sporadic task S models the transmission of a packet stream over a network. In this case, the maximum execution time of all jobs is the time required to transmit a maximum size packet. For simplicity, we let this time be 1. Suppose that the task $S = (1, p, \bar{p}, I)$ also satisfies the $(1/\bar{p}, I/\bar{p})$ ***leaky bucket constraint*.**

In this case, the largest response time occurs when the leaky bucket is full at the beginning t_0 of a busy interval of the server. Starting from t_0, one job with execution time 1 is removed from the bucket every p units of time, while it is being filled at the rate of one job every p units of time. At the time the Xth job is ready for transmission, the bucket is empty.

In other words,

$$X = \frac{I}{\overline{p}} + \left\lfloor \frac{(X-1)p}{\overline{p}} \right\rfloor \leq \frac{I + (X-1)p}{\overline{p}}$$

which gives us

$$X \leq \frac{I - p}{\overline{p} - p}$$

The above observations allow us to conclude that the maximum response time of all jobs in S is bounded from above by

$$W = \left(X + 1 - \frac{(X-1)p}{\overline{p}} \right) \overline{p} = I + \overline{p} \qquad (7.17)$$

We observe that the maximum response time W of a leaky bucket constrained sporadic task is independent of the minimum interarrival time p of jobs. The derivation of the bound in Eq. (7.16) relies only on the fact that the server can complete each job within p units of time after the job reaches the head of the queue of the sporadic task.

$I + \bar{p}$ is also an upper bound on the response times of all jobs in a sporadic task S that meets the $(1/\bar{p}, I/\bar{p})$ leaky bucket constraint and is executed by a schedulable deferrable or sporadic server $(\bar{p}, e)$ in a fixed-priority system.

7.9 A TWO-LEVEL SCHEME FOR INTEGRATED SCHEDULING

This section describes a two-level scheduling scheme that provides timing isolation to individual applications executed on one processor. Each application contains an arbitrary number and types of tasks. By design, the two-level scheme allows different applications to use different scheduling algorithms. So, each application can be scheduled in a best way. The schedulability and real-time performance of each application can be determined independently of other applications executed on the same processor.

7.9.1 Overview and Terminology

According to the two-level scheme, each application is executed by a server. The scheduler at the lower level is called the ***OS scheduler***. It replenishes the budget and sets the deadline of each server in the manners described below and schedules the ready servers on the EDF basis.

At the higher level, each server has a ***server scheduler***; this scheduler schedules the jobs in the application executed by the server according to the algorithm chosen for the application.

Required Capability. In the description below, we use $\mathbf{T}_i$ for $i = 1, 2, \ldots, n$ to denote n real-time applications on a processor; each of these applications is executed by a server. To determine the schedulability and performance of each application $\mathbf{T}_i$, we examine the tasks in it as if the application executes alone on a slower processor whose speed is a fraction s of the speed of the physical processor.

The minimum fraction of speed at which the application is schedulable is called its ***required capacity*** s_i. The required capacity of every application must be less than 1.

For example, the required capacity of an application that contains two periodic tasks $(2, 0.5)$ and $(5, 1)$ is 0.5 if it is scheduled rate-monotonically. The reason is that we can multiple the execution time of each task by 2 and the resultant tasks $(2, 1.0)$ and $(5, 2)$ are schedulable, but if the multiplication factor were bigger, the resultant tasks would not be schedulable. If these tasks are scheduled on the EDF basis, its required capacity is 0.45.

The two-level scheduler creates for all the nonreal-time applications a virtual time-shared processor of speed $\tilde{u}_0$.

Predictable versus Nonpredictable Applications. The OS scheduler needs an estimate of the occurrence time of every event of the application that may trigger a context switch within the application. Such events include the releases and completions of jobs and their requests and releases of resources.

At any time t , the ***next event*** of application $\mathbf{T}_i$ is the one that would have the earliest occurrence time after t among all events of $\mathbf{T}_i$ if the application were to execute alone on a slow processor with speed s_i equal to its required capacity. We call an application that is scheduled according to a preemptive, priority-driven algorithm and contains aperiodic and sporadic tasks and/or periodic tasks with release-time jitters an ***unpredictable application***.

All other types of applications are ***predictable***. An application that contains only periodic tasks with fixed release times and known resource request times is predictable because its server scheduler can compute accurately the occurrence times of its future events. All time-driven applications are predictable because scheduling decisions are triggered by clock interrupts which occur at known time instants. We say that nonpreemptively scheduled applications are predictable.

7.9.2 Scheduling Predictable Applications

An important property of all types of predictable applications is that such an application is schedulable according to the two-level scheme if the size of its server is equal to its requiredcapability and its server is schedulable.

Nonpreemptively Scheduled Applications. Specifically, according to the two-level scheme, each nonpreemptively scheduled application $\mathbf{T}_i$ is executed by a constant utilization server whose size $\tilde{\boldsymbol{u}}_i$ is equal to the required capacity s_i of the application. The server scheduler orders jobs in $\mathbf{T}_i$ according to the algorithm used by $\mathbf{T}_i$.

Let B denote the maximum execution time of nonpreemptable sections of all jobs in all applications in the system and $D_{\min}$ denote the minimum of relative deadlines of all jobs in these applications.

Corollary 7.8 says that all servers are schedulable if the total size of all servers is no greater than $1 - B/D_{\min}$.

We consider a time instant at which the OS scheduler replenishes the server budget and sets the server deadline to execute a job in $\mathbf{T}_i$. (This time instant is the same as the scheduling decision time at which the job would be scheduled if $\mathbf{T}_i$ were executed alone on a slow processor. Since the server is schedulable, it surely will execute for as long as the execution time of the job by the server deadline according to the two-level scheme.)

Applications Scheduled according to a Cyclic Schedule. Each application $\mathbf{T}_i$ that is scheduled according to a cyclic schedule is also executed by a constant utilization server of size equal to s_i . The OS replenishes the budget of the server at the beginning of each frame. At each replenishment, the budget is set to $s_i f$, where f is the length of the frames, and the server deadline is the beginning of the next frame. The server scheduler schedules the application according to its precomputed cyclic schedule.

Preemptively Scheduled Predictable Applications. As with other predictable applications, the size $\tilde{\boldsymbol{u}}_i$ of the server for a predictable application $\mathbf{T}_i$ that is scheduled in a preemptive, priority-driven manner is equal to its required capacity. However, the OS scheduler cannot maintain the server according to the constant utilization/total bandwidth server algorithms. Rather, it replenishes the server budget in the slightly different manner described below.

To motivate the modified replenishment rule, we consider two jobs, J_1 and J_2. The execution time of each is 0.25. Their feasible intervals are *(*0.5*,* 1.5] and *(*0*,* 2], respectively.

The jobs are scheduled preemptively in a priority-driven manner: J_1 has the higher priority and J_2 the lower. These jobs would be schedulable if they were to execute by themselves on a slow processor of speed 0.25. Now, suppose that they are executed by a server of size 0.25 according to the two-level scheme on a processor of speed one. Suppose that the server is maintained according to the constant utilization server or total bandwidth server algorithm: At time 0, the

server is given 0.25 unit of budget, and its deadline is set at 1. The server may execute for 0.25 units of time and complete J_2 by time 0.5 when J_1 is released. When the server budget is replenished again, the deadline for consuming the budget is 2. J_1 may complete as late as time 2 and, hence, may miss its deadline.

If the server for a preemptive, priority-driven application were maintained according to the constant utilization/total bandwidth server algorithms, we could guarantee the schedulability of the application only if the size of its server is 1, even when the required capacity of the application is much smaller than 1.
From time 0 to 0.5, the server is given 0.125 unit of budget more than what is required to complete the portion of J_2.

By giving the server a budget equal to the execution time or the remaining execution time of the job at the head of the server's ready queue, the OS scheduler may introduce a ***form of priority inversion***: A lower-priority job is able to make more progress according to the two-level schedule at the expense of some higher-priority job.

We say that this priority inversion is due to the overreplenishment of the server budget. A way to prevent this kind of priority inversion works as follows.
At each replenishment time of the server for a preemptively scheduled application $\mathbf{T}_i$, let t be the current server deadline or the current time, whichever is later. Let $t\_$ be the next release-time of $\mathbf{T}i$, that is, t' is the earliest release time after t of all jobs in $\mathbf{T}_i$. t' is computed by the server scheduler;

Knowing that a preemption within $\mathbf{T}_i$ may occur at t', the OS scheduler sets the server budget to $\min(e_r, (t' - t)\ \tilde{u}_i)$, where e_r is the execution time of the remaining portion of the job at the head of the server queue, and sets the deadline of the server at $\min(t + e_r / \tilde{u}_i, t')$. This condition, together with conditions that the server size is equal to the required capacity and the total size of all servers is no greater than $1 - B/D_{\min}$, gives a sufficient condition for $\mathbf{T}_i$ to be schedulable when executed according to the two-level scheme.

7.9.3 Scheduling Nonpredictable Applications
In general, the actual release-times of some jobs in a nonpredictable application $\mathbf{T}_i$ may be unknown. It is impossible for its server scheduler to determine the next release-time t' of $\mathbf{T}_i$ precisely. Therefore, some priority inversion due to overreplenishment of the server budget is unavoidable. Fortunately, the bad effect on the schedulability of $\mathbf{T}_i$ can be compensated by making the size $\tilde{\boldsymbol{u}}_i$ of the server larger than the required capacity si of the application.

Budget Replenishment. Let t'_e denote an estimate of the next release time t' of $\mathbf{T}_i$. The server scheduler computes this estimate at each replenishment time t of the server as follows. If the earliest possible release time of every job in $\mathbf{T}i$ is known, the server scheduler uses as t'_e the minimum of the future earliest release times plus an error factor $\varepsilon > 0$.

When the earliest release times of some jobs in $\mathbf{T}_i$ are unknown, t'_e is equal to the $t + \varepsilon$, where t is the current time. After obtaining the value t'_e from the server scheduler, the OS scheduler sets the server budget to $\min(e_r, (t'_e - t)\ \tilde{u}_i)$ and the server deadline at $\min(t + e_r / \tilde{u}_i, t'_e)$.
$\boldsymbol{\varepsilon}$ is a design parameter. It is called the ***quantum size*** of the two-level scheduler.

Required Server Size. The length of priority inversion due to the overreplenishment of the server budget is at most $\boldsymbol{\varepsilon}\tilde{\boldsymbol{u}}_i$. Let $D_{i,\min}$ be the minimum relative deadline of all jobs in $\mathbf{T}_i$. If the size of the server for $\mathbf{T}_i$ satisfies the inequality

$s_i + \varepsilon\ \tilde{u}_i / D_{i,\min} \le \tilde{u}_i$, any higher-priority job that may suffer this amount of blocking can still complete in time. Rewriting this inequality and keeping the equality sign, we get

$$\tilde{u}_i = s_i \frac{D_{i,\min}}{D_{i,\min} - \varepsilon}$$