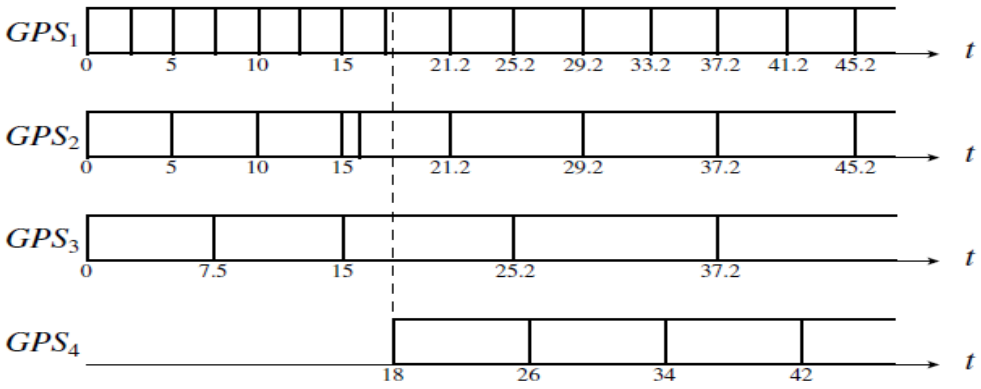
• Before time 18, server GPS_4 being idle, the backlogged servers share the processor proportionally. In other words, the servers attain 2/5, 1/5, and 2/5 of available time, and each of their jobs completes in 2.5, 5, and 7.5 units of time, respectively.

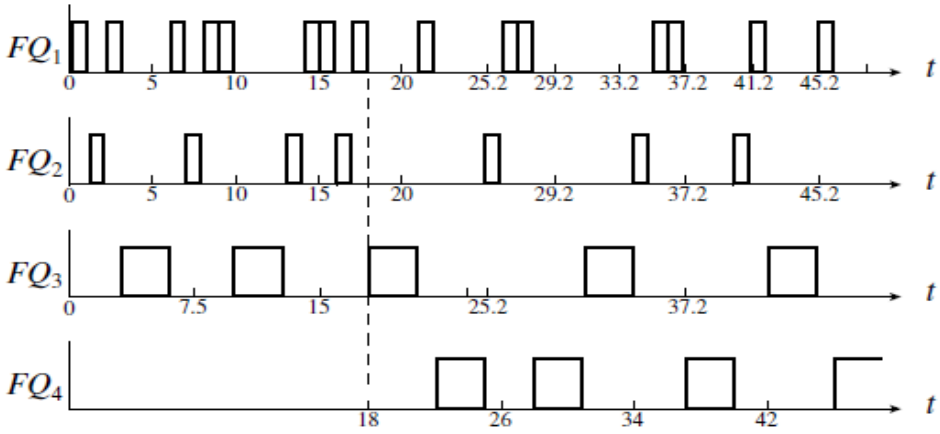
• By time 18, the eighth, fourth, and third jobs of A_1, A_2 and A_3 are at the heads of the queues of the backlogged servers, and their remaining execution times are 0.8, 0.4, and 1.8 respectively.

• Starting from 18, all four servers being backlogged, each server now attains 1/4, 1/8, 1/4, and 3/8 of available time, respectively. This is why the first three servers take an additional 3.2, 3.2, and 7.2 units of time to complete the jobs at the heads of their respective queues. (These jobs are completed at 21.2, 21.2, and 25.2, respectively.) Afterwards, each server GPS_i completes each job in A_i in 4, 8, 12, and 8, units of time, for i = 1, 2, 3, 4, respectively.



(c) Behavior of generalized processor sharing servers.

FIGURE 7–14 (continued)



(d) Behavior of weighted fair-queuing servers with deadlines given in real time.

FIGURE 7–14 (continued)

Figure 7–14(d) gives the corresponding WFQ schedule. The budget of each WFQ server (called FQ_i in the figure) is replenished in the same manner as the corresponding total bandwidth server, except for the way the server deadline is computed. Specifically, we note the following.

• Before time 18, the backlogged WFQ servers behave just like total bandwidth servers, except each of the backlogged servers gets 2/5, 1/5, and 2/5 of processor time, respectively.

Hence whenever a WFQ server FQ_i completes a job, the scheduler gives the server 1, 1, or 3 units of budget and sets its deadline at its current deadline plus 2.5 (i.e., 1 × 5/2), 5 (i.e., 1 × 5/1), or 7.5 (i.e., 3 × 5/2) for i = 1, 2, 3, respectively.

• Immediately before time 18, the three backlogged servers have completed 8, 4, and 2 jobs and their deadlines are at 22.5, 25, and 22.5, respectively.
• At time 18 when FQ_4 also becomes backlogged, the scheduler recomputes the deadlines of FQ_1, FQ_2 and FQ_3 and make them equal to the completion times of the ninth job, fifth job and third job of corresponding tasks according to the GPS schedule.

Their completion times are 25.2, 29.2, and 25.2, respectively. These are the new deadlines of servers FQ_1, FQ_2, and $FQ3$. Also, at this time, the scheduler gives FQ_4 3 units of budget and sets its deadline at 26. The scheduler then queues the servers according to their new deadlines.

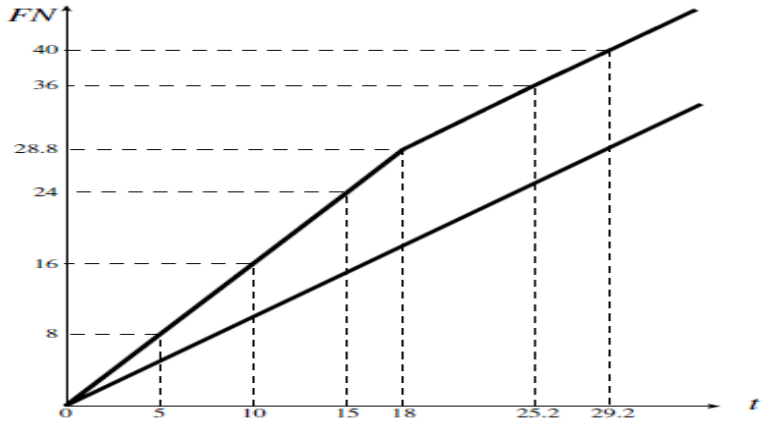
• After time 18, the WFQ servers behave just like total bandwidth servers again.

Virtual Time versus Real-Time. We note that if the scheduler were to replenish server budget it would have to recompute the deadlines of all backlogged servers whenever some server changes from idle to backlogged and vice versa. While this recomputation may be acceptable for CPU scheduling, it is not for scheduling packet transmissions.

A "budget replenishment" in a packet switch corresponds to the scheduler giving a ready packet a time stamp (i.e., a deadline) and inserting the packet in the outgoing queue sorted in order of packet time stamps. To compute a new deadline and time stamp again of an already queued packet would be unacceptable, from the standpoint of both scheduling overhead and switch complexity.

Fortunately, this recomputation of server deadlines is not necessary if instead of giving servers deadlines measured in real time, as we have done in this example, the scheduler gives servers virtual-time deadlines, called ***finish numbers.*** The ***finish number*** of a server gives the number of the round in which the server budget would be exhausted if the backlogged servers were scheduled according to the GPS algorithm.

Figure 7–14(e) shows how finish numbers are related to time for our example. Before time 18, the total size of backlogged servers is only 5/8. If the tasks were executed according to the GPS algorithm, the length of each round would be only 5/8 of what the round length would be when the total size of backlogged servers is 1. In other words, the finish number of the system increases at the rate of 8/5 per unit time. So, at time 2.5, the system just finishes round 8 (= 2.5×8/5); at time 5, the system just finishes round 8, and at time 18, the system is in round 28.8 (= 18 × 8/5). After 18, the total size of backlogged servers is 1. Consequently, the finish number of the system increases at the rate of 1 per unit time.



(e) mapping from real-time finish number (virtual time).

FIGURE 7–14 (continued)

From this argument, we can see that an alternative is for the scheduler to give each server a finish number each time it replenishes the server budget. It then schedules eligible servers according to their finish numbers; the smaller the finish number, the higher the priority. If the scheduler were to use this alternative in our example, the finish numbers of the servers FQ_i , for i = 1, 2, 3, would be the numbers under the time lines of TB_i , respectively, in Figure 7–14(a).

At time 18, the system is in round 28.8, and it takes 8 rounds to complete each job in A_4. Hence, at time 18, the scheduler sets the finish number of FQ_4 to 36.8. At this time, the finish numbers of FQ_1, FQ_2 and FQ_3 are 36, 40, 36, respectively. If we were construct a schedule according to these finish numbers, we would get the schedule in Figure 7–14(d).

Rules of Preemptive Weighted Fair-Queueing Algorithm.

The scheduling and budget consumption rules of a WFQ server are esentially the same as those of a total bandwidth server.

***Scheduling Rule*:** A WFQ server is ready for execution when it has budget and a finish time. The scheduler assigns priorities to readyWFQ servers based their finish numbers: the smaller the finish number, the higher the priority.

***Consumption Rule*:** A WFQ server consumes its budget only when it executes.

In addition to these rules, the weighted fair-queueing algorithm is defined by rules governing the update of the total size of backlogged servers and the finish number of the system and the replenishment of server budget. In the statement of these rules, we use the following notations:

- t denotes the current time, except now we measure this time from the start of the current system busy interval.
- f_n_i denotes the finish number of the server FQ_i , e_i its budget, and $\tilde{u}_i$ its size. e denotes the execution time of the job at the head of the server queue.
- U_b denotes the total size of all backlogged servers at t, and F_N denotes the finish number of system at time t . t_{-1} denotes the previous time when F_N and U_b were updated.

Finally, the system contains n servers whose total size is no greater than one.

Initialization Rules

Initialization Rules

I1 For as long as all servers (and hence the system) are idle, $F\_N = 0$, $U_b = 0$, and $t_{-1} = 0$. The budget and finish numbers of every server are 0.

I2 When the first job with execution time e arrives at the queue of some server FQ_k and starts a busy interval of the system,

(a) $t_{-1} = t$, and increment U_b by $\tilde{u}_k$, and

(b) set the budget e_k of FQ_k to e and its finish number f_n_k to $e/\tilde{u}_k$.

Rules for Updating F_N and Replenishing Budget of FQ_i during a System Busy Interval

R1 When a job arrives at the queue of FQ_i, if FQ_i was idle immediately prior to this arrival,

(a) increment system finish number F_N by $(t - t_{-1})/U_b$,

(b) $t_{-1} = t$, and increment U_b by $\tilde{u}_i$, and

(c) set budget e_i of FQ_i to e and its finish number f_n_i to $F\_N + e/\tilde{u}_i$ and place the server in the ready server queue in order of nonincreasing finish numbers.

R2 Whenever FQ_i completes a job, remove the job from the queue of FQ_i,

(a) if the server remains backlogged, set server budget e_i to e and increment its finish number by $e/\tilde{u}_i$.

(b) if the server becomes idle, update U_b and F_N as follows:

i. Increment the system finish number F_N by $(t - t_{-1})/U_b$,

ii. $t_{-1} = t$ and decrement U_b by $\tilde{u}_i$.

Suppose that in our example in Figure 7–14(d), server FQ_1 were to become idle at time 37 and later at time 55 become backlogged again.

• At time 37, t_{-1} is 18, the value of F_N computed at 18 is 28.8, and the U_b in the time interval $(18, 37]$ is 1. Following rule R2b, F_N is incremented by 37 − 18 = 19 and hence becomes 47.8. U_b becomes 3/4 starting from 37, and t_{-1} = 37.

• At time 55 when FQ_1 becomes backlogged again, the new values of F_N, U_b and t_{-1} computed according to rule R1 are 47.8+(55−37)/0.75 = 71.8, 1, and 55, respectively.

Once the system finish number is found when an idle server becomes backlogged, the finish numbers of the server are computed in the same way as the deadlines of a backlogged total bandwidth server.

We conclude by observing that the response time bound achieved by a WFQ server is the same as that achieved by a total bandwidth server. The completion time of every job in a stream of jobs executed by such a server of size $\tilde{u}$ is never later than the completion time of the job when the job stream is executed by a virtual processor of speed $\tilde{u}$ times the speed of the physical processor.

***7.5 SLACK STEALING IN DEADLINE-DRIVEN SYSTEMS**

It is convenient for us to assume that aperiodic jobs are executed by a ***slack stealer***. The slack stealer is ready for execution whenever the aperiodic job queue is nonempty and is suspended when the queue is empty. The scheduler monitors the periodic tasks in order to keep track of the amount of available slack. It gives the slack stealer the highest priority whenever there is slack and the lowest priority whenever there is no slack. When the slack stealer executes, it executes the aperiodic job at the head of the aperiodic job queue.

This kind of slack-stealing algorithm is said to be ***greedy***: The available slack is always used if there is an aperiodic job ready to be executed.

As an example, we consider again the system of two periodic tasks, T_1 = (2.0, 3.5, 1.5) and T_2 = (6.5, 0.5), which we studied earlier in Figure 7–3. Suppose that in addition to the aperiodic job that has execution time 1.7 and is released at 2.8, another aperiodic job with execution time 2.5 is released at time 5.5. We call these jobs A_1 and A_2, respectively. Figure 7–15 shows the operation of a slack stealer.

1. Initially, the slack stealer is suspended because the aperiodic job queue is empty. When A_1 arrives at 2.8, the slack stealer resumes. Because the execution of the last 0.7 units of $J_{1,1}$ can be postponed until time 4.8 (i.e., 5.5−0.7) and T_2 has no ready job at the time, the system has 2 units of slack.

   The slack stealer is given the highest priority. It preempts $J_{1,1}$ and starts to execute A_1. As it executes, the slack of the system is consumed at the rate of 1 per unit time.
2. At time 4.5, A_1 completes. The slack stealer is suspended. The job $J_{1,1}$ resumes and executes to completion on time.
3. At time 5.5, A_2 arrives, and the slack stealer becomes ready again.

At this time, the execution of the second job $J_{1,2}$ of T_1 can be postponed until time 7.5, and the second job $J_{2,2}$ of T_2 can be postponed until 12.5. Hence, the system as a whole has 2.0 units of slack. The slack stealer has the highest priority starting from this time. It executes A_2.

4. At time 7.5, all the slack consumed, the slack stealer is given the lowest priority. $J_{1,2}$ preempts the slack stealer and starts to execute.

5. At time 9, $J_{1,2}$ completes, and the system again has slack. The slack stealer now has the highest priority. It continues to execute A_2.

6. When A_2 completes, the slack stealer is suspended again. For as long as there is no job in the aperiodic job queue, the periodic tasks execute on the EDF basis.

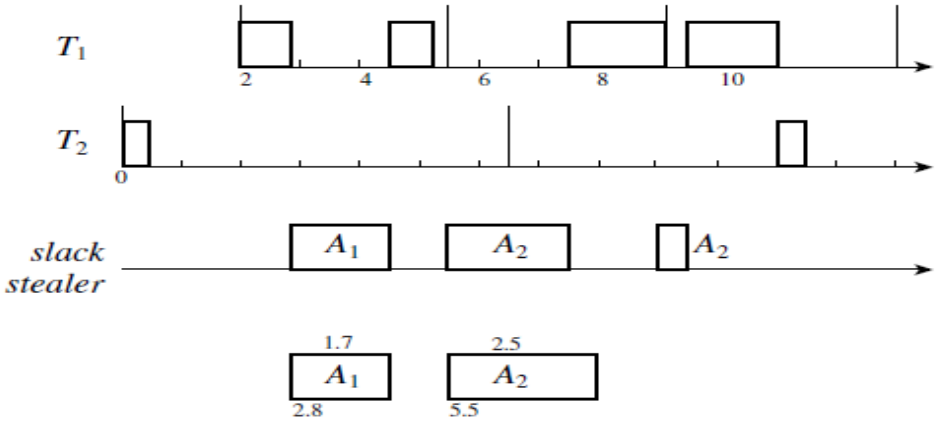


FIGURE 7–15 Example ilustrating a slack stealer in a deadline-driven system: $T_1 = (2.0, 3.5, 1.5)$ and $T_2 = (6.5, 0.5)$.

A slack computation algorithm is ***correct*** if it never says that the system has slack when the system does not, since doing so may cause a periodic job to complete too late. An ***optimal slack computation algorithm*** gives the exact amount of slack the system has at the time of the computation; hence, it is correct. A correct slack computation algorithm that is not optimal gives a lower bound to the available slack.

There are **two approaches to slack computation:** *static* and *dynamic*.

The method used to compute slack in a clock-driven system exemplifies the ***static approach***. According to this approach, the initial slacks of all periodic jobs are computed off-line based on the given parameters of the periodic tasks. The scheduler only needs to update the slack information during run time to keep the information current, rather than having to generate the information from scratch. Consequently, the run-time overhead of the static approach is lower.

A serious limitation of this approach is that the jitters in the release times of the periodic jobs must be negligibly small.We will show later that the slack computed based on the precomputed information may become incorrect when the actual release-times of periodic jobs differ from the release times used to generate the information.

According to the ***dynamic approach***, the scheduler computes the amount of available slack during run time. When the interrelease times of periodic jobs vary widely, dynamic-slack computation is the only choice. The obvious disadvantage of the dynamic-slack computation is its high run-time overhead. However, it has many advantages.

For example, the scheduler can integrate dynamic-slack computation with the reclaiming of processor time not used by periodic tasks and the handling of task overruns. This can be done by keeping track of the cumulative unused processor time and overrun time and taking these factors into account in slack computation.

7.5.1 Static-Slack Computation

To give us some insight into the complexity of slack computation, let us look at the system containing two periodic tasks: $T_1 = (4, 2)$ and $T_2 = (6, 2.75)$. There is an aperiodic job with execution time 1.0 waiting to be executed at time 0. The previous example gives us the impression that we may be able to determine the available slack at time 0 by examining only the slacks of the two current jobs $J_{1,1}$ and $J_{2,1}$. Since we can postpone the execution of $J_{1,1}$ until time 2, this job has 2 units of slack. Similarly, since only 4.75 units of time before the deadline of $J_{2,1}$ are required to complete $J_{2,1}$ and $J_{1,1}$, $J_{2,1}$ has 1.25 units of slack.

If we were to conclude from these numbers that the system has 1.25 units of slack and execute the aperiodic job to completion, we would cause the later job $J_{1,3}$ to miss its deadline. The reason is that $J_{1,3}$ has only 0.5 unit of slack. Consequently, the system as a whole has only 0.5 unit of slack.

In general, to find the correct amount of slack, we must find the minimum among the slacks of all N jobs in the current hyperperiod. If this computation is done in a brute force manner, its time complexity is $O(N)$. Indeed, this is the complexity of some slack computation algorithms.

We now describe an ***optimal static-slack computation algorithm*** proposed by Tia. The complexity of this algorithm is $O(n)$. The number n of periodic tasks is usually significantly smaller than N. To achieve its low run-time overhead, the algorithm makes use of a precomputed slack table that is $O(N_2)$ in size. ***The key assumption of the algorithm is that the jobs in each periodic task are indeed released periodically*.**

To describe this algorithm, it is more convenient for us to ignore the periodic task to which each job belongs. The individual periodic jobs in a hyperperiod are called J_i for $i = 1, 2, \ldots, N$. The deadline of J_i is d_i . The ith jobs in all the hyperperiods are named J_i .

We use t_c to denote the time of a slack computation and $\sigma_i(t_c)$ to denote the *slack* of the periodic job J_i computed at time t_c. $\sigma_i(t_c)$ is equal to the difference between the total available time in $(t_c, d_i]$ and the total amount of time required to complete J_i and all the jobs that are ready in this interval and have the same or earlier deadlines than J_i .

The ***slack of the system $\sigma(t_c)$*** at time t_c is the minimum of the slacks of all the jobs with deadlines after t_c.

Precomputed Slack Table. We take as the time origin the instant when the system begins to execute. For now, we assume that in the absence of aperiodic jobs, the beginning of every hyperperiod coincides with the beginning of a busy interval of the periodic tasks.

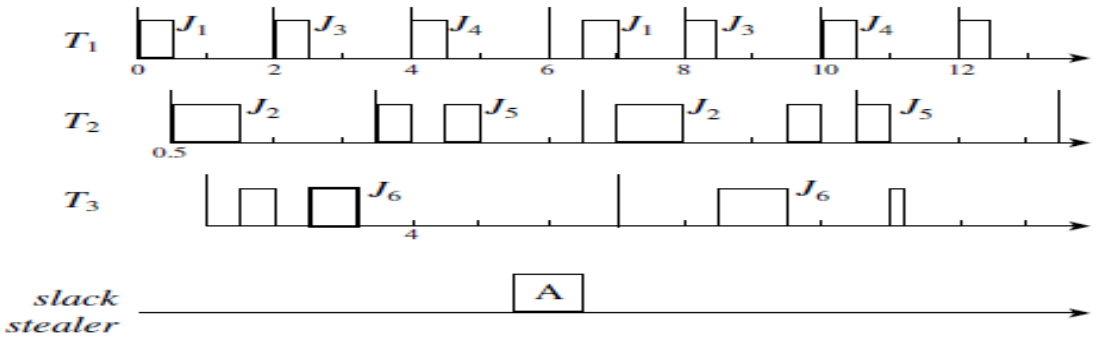
Before the system begins to execute, we compute the initial slack of the N periodic jobs $J_1, J_2, \ldots, J_N$ in each hyperperiod of the periodic tasks. Specifically, the initial slack $\sigma_i(0)$ of the job J_i is given by

$$\sigma_i(0) = d_i - \sum_{d_k \le d_i} e_k \qquad (7.6)$$

Let $\omega(j ; k)$ denote the minimum of all $\sigma_i(0)$ for $i = j, j + 1, \ldots, k - 1, k$. $\omega(j ; k)$ is the ***minimum slack*** of the periodic jobs whose deadlines are in the range $[d_j , d_k]$. Expressed in terms of this notation, the initial slack of the system is $\omega(1; N)$. The initial slack $\sigma_i(0)$ of the job J_i is $\omega(i ; i)$. Rather than storing the $\sigma_i(0)$'s, the precomputed slack table stores the N_2 initial minimum slacks $\omega(j ; k)$ for $1 \le j, k \le N$.

Figure 7–16 gives an example. The system contains three periodic tasks $T_1 = (2, 0.5)$, $T_2 = (0.5, 3, 1)$, and $T_3 = (1, 6, 1.2)$. The relative deadline of every task is equal to its period. An aperiodic job with execution time 1.0 arrives at time 5.5. The figure shows the schedule of the system during the first two hyper periods, as well as the slack table which gives the initial minimum slacks $\omega(j ; k)$ for $j, k = 1, 2, \ldots, 6$.

Dependency of the Current Slack on Past History. Once the system starts to execute, the slack of each individual job changes with time. In particular, the precomputed initial slack of each job J_i does not take into account the events that the processor idles, lower priority jobs execute, and the slack stealer executes. When any of these events occurs before



$W(i,j)$ (i \ j)	1	2	3	4	5	6
1	1.5	1.5	1.5	1.5	1.5	1.5
2		2	2	2	2	2
3			2	2	2	2
4				3.5	3	2.3
5					3	2.3
6						2.3

FIGURE 7–16 Slack table of $T_1 = (2, 0.5)$, $T_2 = (0.5, 3, 1.0)$, and $T_3 = (1, 6, 1.2)$.

d_i and takes away some available time from J_i , the slack of the job becomes smaller than the precomputed value.

For example, J_6 in Figure 7–16 executes for 0.5 unit of time before time 2. The precomputed initial slack of J_3 does not take into account this 0.5 unit of time. At time 2, the slack of J_3 is not equal to the precomputed value 2; rather it is equal to 1.5. At time 6.5, the slack stealer has executed since the beginning of the second hyperperiod for 0.5 unit of time. For this reason, the slacks of all the jobs in the hyperperiod is reduced by 0.5 unit from their respective precomputed values. In general, to facilitate slack computation, the scheduler maintains the following information on the history of the system:

1. the ***total idle time I*** , which is the total length of time the processor idles since the beginning of the current hyperperiod.

2. the ***total stolen time ST*** , which is the time consumed by the slack stealer since the beginning of the current hyperperiod, and

3. the **execution time ξ_k** of the completed portion of each periodic job J_k in the current hyperperiod.

At time t_c of a slack computation, the slack of a job J_i that is in the current hyperperiod and has a deadline after t_c is equal to

$$\sigma_i(t_c) = \sigma_i(0) - I - ST - \sum_{d_i < d_k} \xi_k \qquad (7.7)$$

In the previous example, at time 2, ξ_6 is 0.5, *I* and *ST* are 0. Hence, the slacks of J_3, J_4 and J_5 are reduced by 0.5 unit from their initial values 2.0, 3.5, and 3, respectively. At time 3.5, ξ_6 is 1.2, *I* is 0.3, and *ST* is equal to 0. The slacks of J_4 and J_5 are reduced to 2.0 and 1.5, respectively. If we want to compute the slack of the system at this time, we need to find only the minimum slack of the jobs J_4, J_5, and J_6. When the second hyperperiod begins, the values of *I* , *ST* and ξ_i 's are all 0, and the slack of the system is given by $\omega(1; 6)$, which is 1.5. At time 6.5, *ST* becomes 0.5; the slack of every job in the second hyperperiod is reduced by 0.5 unit, and the slack of the system is reduced to 1.0.

Computing the Current Slack of the System. We can speed up the slack computation by first partitioning all the jobs into *n* disjoint subsets. It suffices for us to compute and examine the slack of a job in each set. By doing so, we can compute the slack of the system in *O(n)* time.

To see why this speedup is possible, we remind ourselves that at t_c, only one job per periodic task is current, and only current jobs could have executed before t_c. To distinguish current jobs from the other jobs in the hyperperiod, we call the current job of T_i J_{ci}, for $i = 1, 2, \ldots, n$. The deadline of J_{ci} is d_{ci} . The current jobs are sorted by the scheduler in nondecreasing order of their deadlines in the priority queue of ready periodic jobs. Without loss of generality, suppose that $d_{c1} < d_{c2} < \cdots < d_{cn}$.

We now partition all the periodic jobs that are in the hyperperiod and have deadlines after *tc* into *n* subsets $\mathbf{Z}_i$ for $i = 1, 2, \ldots, n$ as follows. A job is in the subset $\mathbf{Z}_i$ if its deadline is in the range $[d_{ci}, d_{ci+1})$ for $i = 1, 2, \ldots, n-1$. Hence, $\mathbf{Z}_i$ is $\{J_{ci}, J_{ci+1}, \ldots, J_{ci+1-1}\}$. The subset $\mathbf{Z}_n$ contains all the jobs in the current hyperperiod whose deadlines are equal to or larger than d_{cn} .

This partition is illustrated by Figure 7–17. The tick marks show the deadlines of the current jobs and of the jobs in each of the subsets. The job that has the latest deadline among all the jobs in $\mathbf{Z}_i$ is $J_{ci+1}-1$. This partition can be done in *O(n)* time since the jobs are presorted and we know the index of every job.

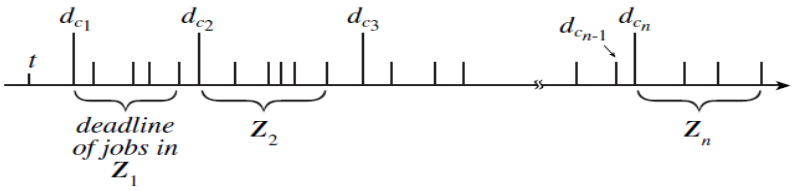


FIGURE 7–17 Partitioning of all jobs into *n* subsets.

The values of ξ_k 's are nonzero only for current jobs. From this observation and Eq. (7.7), we can conclude that ***in the computation of the current slacks of all the jobs in each of the subset Zi , the amounts subtracted from their respective initial slacks are the same*.**

Let us suppose that we want to compute the slack of the system in Figure 7–16 at time 1.75. Both total idle time *I* and stolen time *ST* are zero. The current jobs at the time are J_1, J_2, and J_6, and the execution times of their completed

portions are 0.5, 1.0, and 0.25, respectively. The deadlines of these three jobs partition the six jobs in this hyperperiod into three subsets: $\mathbf{Z}_1 = \{J_1\}$, $\mathbf{Z}_2 = \{J_2, J_3, J_4, J_5\}$, and $\mathbf{Z}_3 = \{J_6\}$. We need not be concerned with $J1$ and $J2$ since they are already completed at time 1.75. The slacks of the other jobs in $\mathbf{Z}_2$ are equal to their respective initial slacks minus 0.25, the execution time of the completed portion of the current job J_6, because d_6 is later than their own deadlines.

On the other hand, the slack of the J_6, the only job in $\mathbf{Z}_3$, is equal to its initial slack because there is no current job whose deadline is later than d_6. Since the slack of every job in each subset $\mathbf{Z}_i$ is equal to its initial slack minus the same amount as all the jobs in the subset, the job that has the smallest initial slack also has the smallest current slack. In other words, the minimum slack of all the jobs in $\mathbf{Z}_i$ is

$$\omega_i(t_c) = \omega(c_i; c_{i+1} - 1) - I - ST - \sum_{k=i+1}^{n} \xi_{c_k} \tag{7.8a}$$

for $i = 1, 2, \ldots, n-1$, and

$$\omega_n(t_c) = \omega(c_n; N) - I - ST \tag{7.8b}$$

The slack $\sigma(t_c)$ of the system is given by

$$\sigma(t_c) = \min_{1 \le i \le n} \omega_i(t_c) \tag{7.8c}$$

The scheduler can keep the values of I , ST and ξ_{ci} 's up to date by using n +2 registers, one for each of these variables, together with a counter. These variables are set to 0 at the beginning of each hyperperiod. The counter is loaded with the current value of total idle time whenever the processor becomes idle and is incremented as long as the processor idles.

Precomputed Slack Table: $\omega(i; k)$ gives the minimum slack of all periodic jobs with deadlines in the range $[d_i, d_k]$ for $i, k = 1, 2, \ldots, N$.

Slack Computation at time t

1. Obtain from the ready periodic job queue the sorted list of current jobs J_{c_i} for $i = 1, 2, \ldots, n$.
2. Partition the periodic jobs that are in the current hyperperiod and have deadlines after t into n subsets such that the subset $\mathbf{Z}_i$ contains all the jobs with deadlines equal to or larger than d_{c_i} but less than $d_{c_{i+1}}$.
3. Compute the slack $\sigma(t)$ of the system according to Eq. (7.8).

Operations of the Scheduler

- Initialization:
  - Create a slack stealer and suspend it.
  - Set I, ST, and ξ_i for $i = 1, 2, \ldots, N$ to 0 at the beginning of each hyperperiod.
- Schedule periodic jobs in the EDF order as follows: Whenever a job is released or complete or when the slack is exhausted do the following:
  - Update I, ST, or ξ_i for $i = 1, 2, \ldots, N$.
  - If the aperiodic job queue is nonempty,
    - Do slack computation to find $\sigma(t)$.
    - If $\sigma(t) > 0$, then schedule the slack stealer.
      Else, schedule the highest priority periodic job.
    Else, schedule the highest priority periodic job.

FIGURE 7–18 Operations of an EDF scheduler to accommodate a slack stealer.

The content of the counter replaces the old value of I when the processor starts to execute again. Similarly, the counter keeps tracks of the amount of time spent executing the slack stealer or a periodic job each time it executes. The value

of *ST* is updated at each time instant when the aperiodic job queue becomes empty or when the slack stealer is preempted. The execution time ξ_{ci} of the completed portion of each current job J_{ci} is updated whenever the job completes or is preempted, as well as when the slack of the system needs to be computed.

The pseudocode description in Figure 7–18 summarizes the operation of a scheduler in a deadline-driven system where aperiodic jobs are executed by a slack stealer. The description does not include the actions to place ready jobs in the appropriate queues in priority order.

7.5.2 Practical Considerations

Effect of Phases. When the phases of periodic tasks are arbitrary, the end of the first hyperperiod may not be the end of a busy interval. In this case, even in the absence of aperiodic jobs, the schedule of the second hyperperiod may not be the same as that of the first hyperperiod.

As an example, we consider a system of two tasks, $T_1 = (2, 1)$ and $T_2 = (1, 3, 1.5)$. The length of a hyperperiod is 6 and the number N of jobs in each hyperperiod is five. If we were to compute the slack of the system based on the precomputed slacks of the first five jobs, we would conclude that the system has 0.5 unit of slack in each hyperperiod.

When the periodic tasks are not in phase, we need to determine whether the end of the first hyperperiod is also the end of a busy interval of the periodic tasks in the absence of aperiodic jobs. If it is, as exemplified by the system in Figure 7–16, the schedule of the periodic task system in the absence of aperiodic jobs and release-time jitters is cyclic with period H from the start. It suffices for us to precompute the slacks of the first N jobs.

If the end of the first hyperperiod is in the midst of a busy interval, the schedule of the periodic tasks has an initial transient segment which ends when the first busy interval ends and then is cyclic with period H. Therefore, we need to precompute the slacks of periodic jobs that execute in the first busy interval and then the slacks of the N jobs in the hyperperiod that begins when the second busy interval begins.

The slack of the system should be computed based on the latter N entries once the second busy interval begins. In the previous example, the first busy intervals ends at 3.5. Jobs J_1 and J_3 of T_1 and J_2 of T_2 execute in the first busy interval. After time 4.0, we compute the system slack based on the precomputed slacks of jobs $J_4, \ldots, J_8$.

Effect of Release-Time Jitters. Release-time jitters are often unavoidable and the periodic task model and priority-driven scheduling algorithms allow for the jitters. A critical question is whether the results of a static-slack computation based on the assumption that the jobs in each task are released periodically remain correct when the interrelease times of jobs in each task may be larger than the period of the task. An examination of the example in Figure 7–16 can give us some insight into this question.

Suppose that the release times of J_3 and J_4 are delayed to 2.1 and 4.3, respectively, while the other jobs in this hyperperiod are released at their nominal release times. The initial slacks of the jobs J_1, J_3, and J_4 given by the precomputed slack table are 1.5, 2.0, and 3.5. Since the actual deadline of the J_1 remains to be 2.0, its initial slack is still 1.5. However, the actual deadlines of J_3 and J_4 are 4.1 and 6.3, respectively. Therefore, their actual initial slacks are 2.1 and 3.8, respectively. Similarly, if the release time of $J5$ is delayed by a small amount, say by 0.1 unit, the actual initial slack of this job is 0.1 unit more than is given by the precomputed slack table.

The precomputed initial slacks of other jobs remain accurate despite these late releases. In this case, the precomputed initial slacks give us lower bounds to the actual initial slacks.In general, as long as the jitters in the release times are so small that priorities of all the jobs are not changed as a consequence Eq. (7.8) continues to be correct. However, when the release-time jitters are large compared to the separations between deadlines, the actual order and the priorities of the jobs may differ from their precomputed values.

7.5.3 Dynamic-Slack Computation

In the presence of release-time jitters, the slack of a system may have to be computed dynamically during run time. We focus here on an algorithm that computes a lower bound on the slack of the system at time t_c when the scheduler needs the bound, without relying on any precomputed slack information. So, the adjectives "current" and "next" are relative with respect to the slack computation time t_c.

The schedule segments during different busy intervals of the periodic tasks are independent of each other. Therefore, when computing the slack $\sigma(t_c)$, it is safe to consider only the periodic jobs that execute in the current busy interval, provided that any available slack in subsequent busy intervals is not included in $\sigma(t_c)$.

Information Maintained by the Scheduler. We let ξ_{ci} denote the execution time of the completed portion of the current job J_{ci} of the periodic task T_i . To support dynamic slack computation, the scheduler keeps up to date the value of ξ_{ci} for each periodic task T_i . Let $\tilde{\varphi}_i$ denote the earliest possible release time of the next job in T_i . The scheduler can keep the value of $\tilde{\varphi}_i$ up-to-date by adding p_i to the actual release time of the current job in T_i when the job is released.

Estimated Busy Interval Length. The slack computation is done in two steps.

In the first step, the scheduler computes an upper bound on the length X of time to the end of the current busy interval of the periodic tasks. The length X is the longest when the jobs in each task T_i are released periodically with period pi . Under this condition, the maximum processor-time demand $w(x)$ of all periodic jobs during the interval $(t_c, t_c + x)$ is given by

$$w(x) = \sum_{i=1}^{n} (e_i - \xi_i) + \sum_{i=1}^{n} \left\lceil \frac{(t_c + x - \tilde{\phi}_i) u_{-1}(t_c + x - \tilde{\phi}_i)}{p_i} \right\rceil e_i$$

where $u{-}1(t)$ denotes a unit step function which is equal to zero for $t < 0$ and is equal to one when $t \geq 0$.

The first sum on the right-hand side of this equation gives the total remaining execution time of all the current periodic jobs at time t_c. The second sum gives the total maximum execution time of all the jobs that are released in $(t_c, t_c + x)$. The length X is equal to the minimum of the solutions of $w(x) = x$ and can be found by solving this equation iteratively.

The current busy interval ends at time $END = t_c + X$. Let $BEGIN$ be the earliest possible instant at which the next busy interval can begin. Since $\lfloor (END - \tilde{\varphi}i)/pi \rfloor$ jobs of Ti are released after $\tilde{\varphi}_i$ and before END for all i , the earliest possible time $BEGIN$ at which next busy interval can begin is given by

$$BEGIN = \min_{1 \leq i \leq n} \left(\tilde{\phi}_i + \left\lceil \frac{(END - \tilde{\phi}_i) u_{-1}(END - \tilde{\phi}_i)}{p_i} \right\rceil p_i \right)$$

Slack Computation. We can save some time in slack computation by considering only jobs in the current busy interval. Let $\mathbf{J}_c$ denote the subset of all the periodic jobs that execute in the interval (t_c, END). In the second step, the scheduler considers only the jobs in $\mathbf{J}_c$, computes a lower bound on the slack of each of these jobs, and takes as the slack of the system the minimum of the lower bounds.

Specifically, a lower bound on the slack of any job J_i in the set $\mathbf{J}_c$ is the difference between $\min(d_i , BEGIN)$ and the total execution time of all the jobs that are in $\mathbf{J}_c$ and have deadlines equal to or before d_i .

By using this lower bound, the scheduler eliminates the need to examine jobs that are released in the next busy interval. Again, the slack of the system is equal to the minimum of such lower bounds on slacks of all the jobs in $\mathbf{J}_c$. This computation can be done in $O(N_b)$ time where N_b is the maximum number of jobs in a busy interval. In the special case where the relative deadline of every periodic task is equal to its period, *BEGIN − END* is a lower bound to the slack of the system.

*7.6 SLACK STEALING IN FIXED-PRIORITY SYSTEMS

In principle, slack stealing in a fixed-priority system works in the same way as slack stealing in a deadline-driven system. However, both the computation and the usage of the slack are more complicated in fixed-priority systems.

7.6.1 Optimality Criterion and Design Consideration

To illustrate the issues, Tia provided the example in Figure 7–19. The system contains three periodic tasks: $T_1 = (3, 1)$, $T_2 = (4, 1)$, and $T_3 = (6, 1)$. They are scheduled rate monotonically.

If the system were deadline-driven, it would have 2 units of slack in the interval $(0, 3]$, but this system has only 1 unit. The reason is that once $J_{1,2}$ becomes ready, $J_{2,1}$ must wait for it to complete. As a consequence, $J_{2,1}$ must complete by time 3, although its deadline is 4. In essence, 3 is the effective deadline of $J_{2,1}$, and its slack is determined by the effective deadline.

Figure 7–19(a) shows the schedule for the case when the 1 unit of slack is not used before time 3. At time 3, $J_{3,1}$ has already completed. $J_{1,2}$ and $J_{2,2}$ can start as late as time 5 and 7, respectively, and still complete in time. Therefore, the system has two units of slack at time 3.

Figure 7–19(b) shows the schedule for the other case: The 1 unit of slack is used before time 3. $J_{3,1}$ is not yet complete at time 3. Consequently, $J_{1,2}$ and $J_{2,2}$ must execute immediately after they are released, even though their deadlines are 6 and 8; otherwise, $J_{3,1}$ cannot complete in time. Under this condition, the system has no more slack until time 6.

Now suppose that aperiodic jobs A_1 and A_2 arrive at times 2 and 3, respectively, and their execution times are equal to 1. If the slack stealer executes A_1 immediately upon arrival, the job can be completed at 3 and have the minimum response time of 1. Since the system has no more slack until time 6, $A2$ cannot be completed until time 7.

On the other hand, if the scheduler waits until time 3 and then schedules the slack stealer, the aperiodic jobs are completed at times 4 and 5, respectively. A_2 now has the minimum response time of 2. This reduction is achieved at the expense of A_1, whose response time is no longer the minimum possible.

This example points out the following important facts. These facts provide the rationales for the slack-stealing algorithm described below.

1. *No slack-stealing algorithm can minimize the response time of every aperiodic job in a fixed-priority system even when prior knowledge on the arrival times and execution times of aperiodic jobs is available*.

2. The amount of slack a fixed-priority system has in a time interval may depend on when the slack is used. To minimize the response time of an aperiodic job, the decision on when to schedule the job must take into account the execution time of the job.

Because of (1), we use here a weaker optimality criterion: A slack-stealing algorithm for fixed-priority systems is *optimal in the weak sense* if it is correct and it minimizes the response time of the job at the head of the aperiodic job queue. Because of (2), an optimal slack-stealing algorithm for fixed-priority systems does not use available slack greedily. The remainder of this section describes a slack-stealing algorithm.

7.6.2 Static Slack Computation in Fixed-Priority Systems

The previous example pointed out that the slack of each periodic job should be computed based on its effective deadline, not its deadline. The effective deadline of every job in the highest priority periodic task is equal to its deadline.

For $i > 1$, the ***effective deadline*** $de_{i,j}$ of a job $J_{i,j}$ is equal to the deadline $d_{i,j}$ of the job if higher-priority periodic tasks have no ready jobs immediately before $d_{i,j}$.
Otherwise, if $d_{i,j}$ is amid or at the end of a level-π_{i-1} busy interval, $de_{i,j}$ is equal to the beginning of this busy interval.

Just as in deadline-driven systems, a static-slack computation at any time t_c within a hyperperiod begins with the precomputed initial slack $\sigma_{i,j}(0)$ of every periodic job $J_{i,j}$ in the hyperperiod. The initial slack of $J_{i,j}$ is given by

$$\sigma_{i,j}(0) = \max\left(0, d^e_{i,j} - \sum_{k=1}^{i} e_k \left\lceil \frac{d^e_{i,j}}{p_k} \right\rceil\right) \qquad (7.9)$$

The effective deadlines of all N periodic jobs in a hyperperiod can be determined by constructing a schedule of the periodic tasks for the hyperperiod when $\sigma_{i,j}(0)$'s are computed. The N effective deadlines and initial amounts of slack are stored for use at run time. The effective deadlines and initial slacks of jobs in subsequent hyperperiods can easily be computed from the respective stored values of jobs in the first hyperperiod.

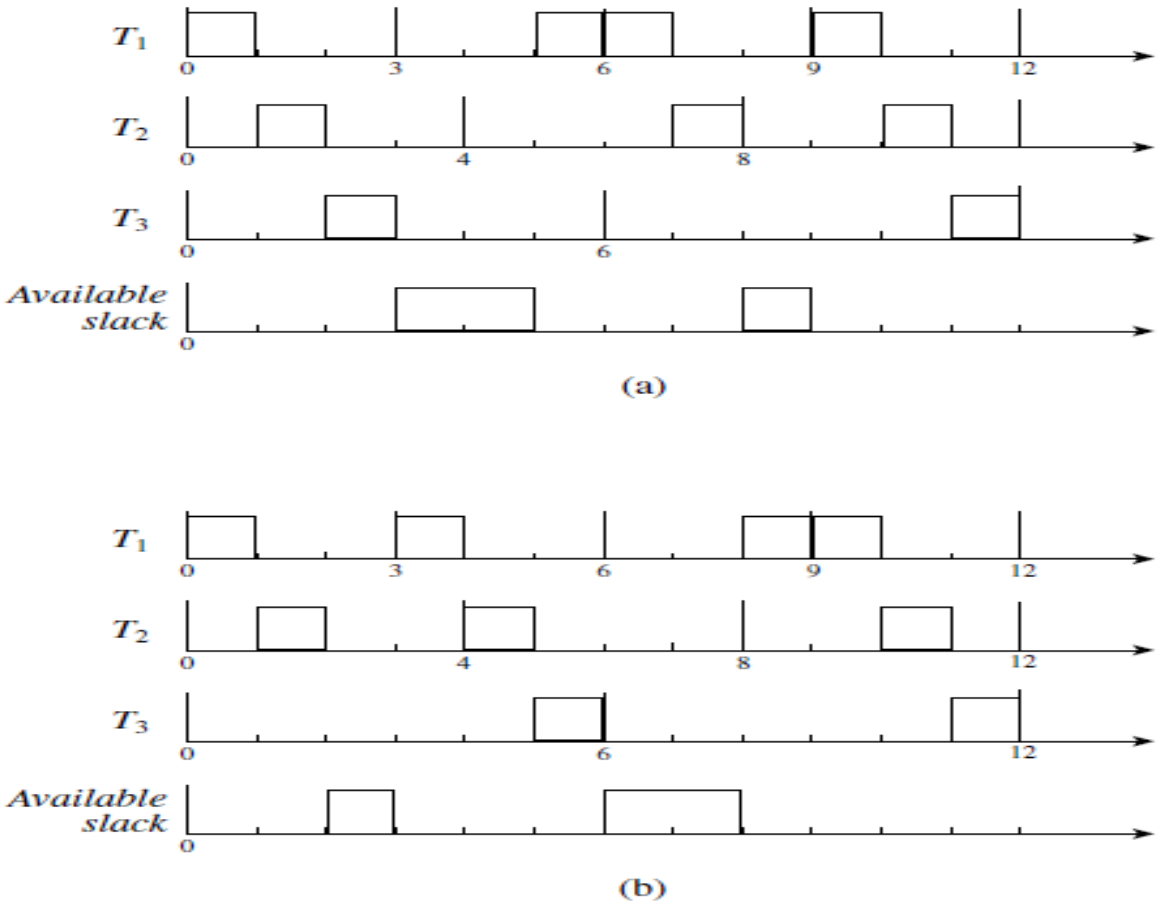


FIGURE 7–19 Example illustrating slack stealing in fixed-priority systems containing $T_1 = (3, 1)$, $T_2 = (4, 1)$, $T_3 = (6, 1)$.

Slack Functions. Table 7–1 lists the notations used in the subsequent description of the slack-stealing algorithm. The scheduler does a slack computation when it needs to decide the priority of the slack stealer. Let t be a future time instant (i.e., $t > t_c$) and $J_{i,j}$ be the first job in T_i whose effective deadline is after t. In other words, $de_{i,j-1} \leq t < de_{i,j}$.

$$\sigma_i(t_c, t) = \begin{cases} \sigma_{i,j}(0) - I - ST - \Xi_i & \text{if } J_{i,j} \text{ is not completed at } t \\ \sigma_{i,j+1}(0) - I - ST - \Xi_i & \text{if } J_{i,j} \text{ is completed at } t \end{cases} \qquad (7.10a)$$

gives the amount of time before $x_i(t)$ that is not needed by jobs in $\mathbf{H}_i(t_c, x(t))$ for them to complete before $x_i(t)$. We call $\sigma_i(t_c, t)$ the ***slack function of T_i***. The ***minimum slack function*** of the periodic tasks

$$\sigma^*(t_c, t) = \min_{1 \leq i \leq n} \sigma_i(t_c, t) \qquad (7.10b)$$

gives the minimum amount of available slack from t_c to $x_i(t)$.

To support the computation of $\sigma_i(t_c, t)$, the scheduler keeps up-to-date the total idle time I and stolen time ST.

TABLE 7–1 Notations Used in the Description of the Slack-Stealing Algorithm

t_c:	the time at which a slack computation is carried out
t_0:	the beginning of the current hyperperiod, the beginning of the first hyperperiod being 0
$d^e_{i,j}$:	the effective deadline of the job $J_{i,j}$
$\sigma_{i,j}(0)$:	precomputed initial slack of the jth job $J_{i,j}$ of T_i in the current hyperperiod
I:	the total idle time, i.e., the cumulative amount of time since t_0 during which the processor idles
ST:	the total stolen time, i.e., the cumulative amount of time since t_0 during which aperiodic jobs execute.
ξ_i, for $i = 1, 2, \ldots, n$:	the cumulative amount of processor time since t_0 used to execute T_i
Ξ_i, for $i = 1, 2, \ldots, n$:	the cumulative amount of time since t_0 used to execute periodic tasks with priorities lower than T_i
$x_i(t)$, for a time instant t in the range $[d^e_{i,j-1}, d^e_{i,j})$:	$x_i(t)$ is equal to the effective deadline $d^e_{i,j}$ of $J_{i,j}$ if the job is not complete by time t, and is equal to $d^e_{i,j+1}$ if $J_{i,j}$ is complete by t.
$\mathbf{T}_{i-1}$:	the subset of tasks, other than T_i, with priorities equal to or higher than T_i.
$\mathbf{H}_i(t_c, t)$, $t > t_c$:	the subset of periodic jobs that are in T_i or $\mathbf{T}_{i-1}$ and are active at some time during the interval $(t_c, t]$—A job is active in the interval between its release time and its effective deadline.
$\sigma_{i,j}(t_c, t)$:	the slack function of T_i
$\sigma^*(t_c, t)$:	the slack function of the system
$y_1 < y_2 < \ldots, < y_k$:	locations of steps of $\sigma^*(t_c, t)$, i.e., the values of t at which $\sigma^*(t_c, t)$ increases
$\sigma^*(y_k)$:	the amount of slack from t_c to y_k
z_i:	the next step of $\sigma_i(t_c, t)$ after t_c and the latest known step of $\sigma^*(t_c, t)$
A:	the aperiodic job at the head of the aperiodic job queue
e_A:	the remaining execution time of A

can do this.) In addition, for each periodic task T_i in the system, the scheduler updates the cumulative amount ξ_i of time during which the processor executes T_i each time when a job in the task completes or is preempted. Because $\Xi_i = \Xi_{i-1} + \xi_{i-1}$, the scheduler can obtain the values of Ξ_i's needed in Eq. (7.10a) by a single pass through the ξ_i's. This takes $O(n)$ time.

To illustrate, we return to the previous example. The effective deadline of every job except $J_{2,1}$ is equal to its deadline. Because $d_{2,1}$ is at the end of a busy interval of T_1 that begins at 3, $d^e_{2,1}$ is 3. At time 0, I, ST, and Ξ_i are equal to 0. Figure 7–20(a) shows the slack functions $\sigma_i(0, t)$ computed at time 0 for $i = 1$, 2, and 3 and t in the first hyperperiod (0, 12). The minimum slack function $\sigma^*(0, t)$ shows that the periodic tasks have 1 unit of slack before 6 and 2 more units after 6. Figure 7–20(b) shows the slack functions computed at time 3 for the case where the slack is not used before 3. At the time, $J_{2,1}$ is complete. According to Eq. (7.10a), the slack function $\sigma_2(3, t)$ for $t \leq 8$ is equal to the initial slack ($= 3$) of $J_{2,2}$ minus Ξ_3 ($= 1$) and hence is 2. Similarly, $J_{3,1}$ is completed. For $t > 3$, the slack function $\sigma_3(3, t)$ is equal to initial slack $\sigma_{3,2}(0) = 3$ of $J_{3,2}$. The amount of slack available before 6 increases by 1 at time 3. This is what we observed earlier from Figure 7–19.