UNIT-IV

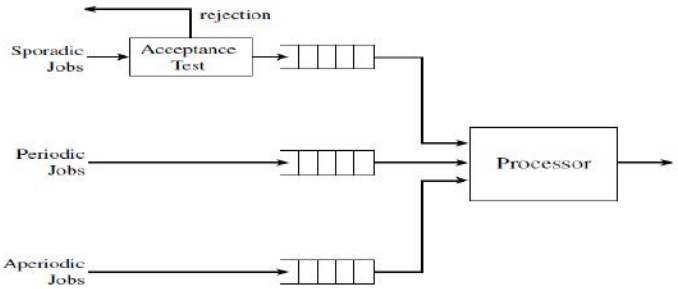
Scheduling Aperiodic and Sporadic Jobs in Priority-Driven Systems

ASSUMPTIONS AND APPROACHES:

Aperiodic and sporadic jobs are also independent of each other and of the periodic tasks. We assume that every job can be preempted at any time. We do not make any assumptions on the inter release-times and execution times of aperiodic jobs. Indeed, most algorithms used to schedule aperiodic jobs do not even require the knowledge of their execution times after they are released.

Objectives, Correctness, and Optimality:

When the periodic tasks are scheduled according to the given algorithm and there are no aperiodic and sporadic jobs, the periodic tasks meet all their deadlines. For the concreteness, we assume that the operating system maintains the priority queues shown in below Figure.



The ready periodic jobs are placed in the periodic task queue, ordered by their priorities that are assigned according to the given periodic task scheduling algorithm. Similarly, each accepted sporadic job is assigned a priority and is placed in a priority queue, which may or may not be the same as the periodic task queue. Each newly arrived aperiodic job is placed in the aperiodic job queue. Moreover, aperiodic jobs are inserted in the aperiodic job queue and newly arrived sporadic jobs are inserted into a waiting queue to await acceptance without the intervention of the scheduler.

The algorithms described in this chapter determine when aperiodic or sporadic jobs are executed. We call them *aperiodic job and sporadic job scheduling algorithms*; they are solutions to the following problems:

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

2. The scheduler tries to complete each aperiodic job as soon as possible. The problem is how to do so without causing periodic tasks and accepted sporadic jobs to miss their deadlines.

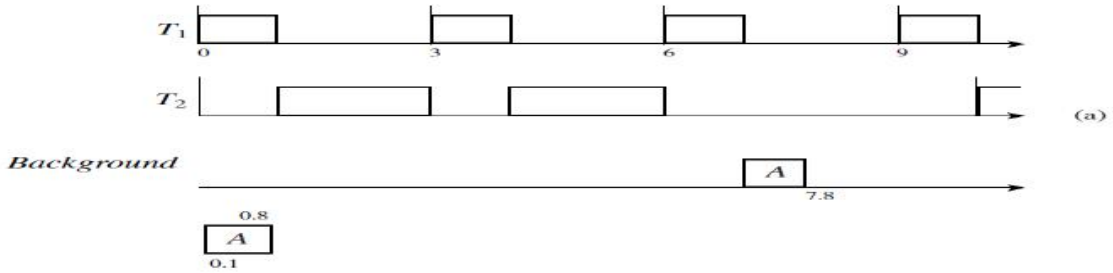
Alternative Approaches:

All the algorithms described in this chapter attempt to provide improved performance over the three commonly used approaches. These approaches are background, polled, and interrupt driven executions.

Background and Interrupt-Driven Execution versus Slack Stealing:

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

Example: we consider the system of two periodic tasks $T1 = (3, 1)$ and $T1 = (10, 4)$ .The tasks are scheduled rate monotonically. Suppose that an aperiodic job A with execution time equal to 0.8 is released (i.e., arrives) at time 0.1. If this job is executed in the background, its execution begins after $J1,3$ completes (i.e., at time 7) as shown in Figure (a). Consequently, its response time is 7.7.

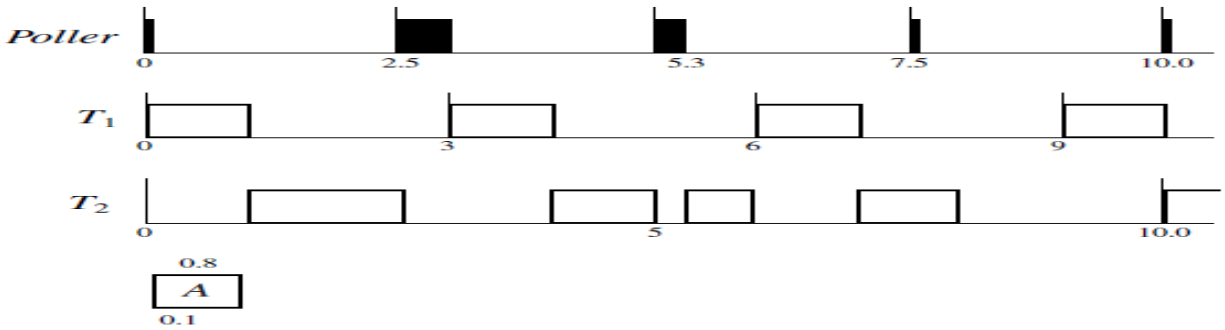


An obvious way to make the response times of aperiodic jobs as short as possible is to make their execution interrupt-driven. Whenever an aperiodic job arrives, the execution of periodic tasks are interrupted, and the aperiodic job is executed. In this example, A would execute starting from 0.1 and have the shortest possible response time. The problem with this scheme is equally obvious: If aperiodic jobs always execute as soon as possible, periodic tasks may miss some deadlines.

The obvious solution is to postpone the execution of periodic tasks only when it is safe to do so.Algorithms that make use of the available slack times of periodic and sporadic jobs to complete aperiodic jobs early are called *slack-stealing algorithms*

Polled Executions versus Bandwidth-Preserving Servers:

Polling is another commonly used way to execute aperiodic jobs. In our terminology, a *poller* or *polling server* (p_s , e_s) is a periodic task: p_s is its polling period, and e_s is its execution time. The poller is ready for execution periodically at integer multiples of p_s and is scheduled together with the periodic tasks in the system according to the given priority-driven algorithm. When it executes, it examines the aperiodic job queue. If the queue is nonempty, the poller executes the job at the head of the queue. The poller suspends its execution or is suspended by the scheduler either when it has executed for e_s units of time in the period or when the aperiodic job queue becomes empty, whichever occurs sooner. It is ready for execution again at the beginning of the next polling period. On the other hand, if at the beginning of a polling period the poller finds the aperiodic job queue empty, it suspends immediately. It will not be ready for execution and able to examine the queue again until the next polling period.



DEFERRABLE SERVERS

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

Operations of Deferrable Servers

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

Consumption Rule:

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

Replenishment Rule:

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

The server is not allowed to cumulate its budget from period to period. Stated in another way, any budget held by the server immediately before each replenishment time is lost.

Example: Figure (a) shows that the deferrable server *TDS* = *(*3, 1*)* has the highest priority. The periodic tasks *T*1 = *(*2.0, 3.5, 1.5*)* and *T*2 = *(*6.5, 0.5*)* and the server are scheduled rate-monotonically. Suppose that an aperiodic job *A* with execution time 1.7 arrives at time 2.8.

1. At time 0, the server is given 1 unit of budget. The budget stays at 1 until time 2.8.When *A* arrives, the deferrable server executes the job. Its budget decreases as it executes.

2. Immediately before the replenishment time 3.0, its budget is equal to 0.8. This 0.8 unit is lost at time 3.0, but the server acquires a new unit of budget. Hence, the server continues to execute.

3. At time 4.0, its budget is exhausted. The server is suspended, and the aperiodic job *A* waits.

4. At time 6.0, its budget replenished, the server resumes to execute *A*.

5. At time 6.5, job *A* completes. The server still has 0.5 unit of budget. Since no aperiodic job waits in the queue, the server suspends itself holding this budget.

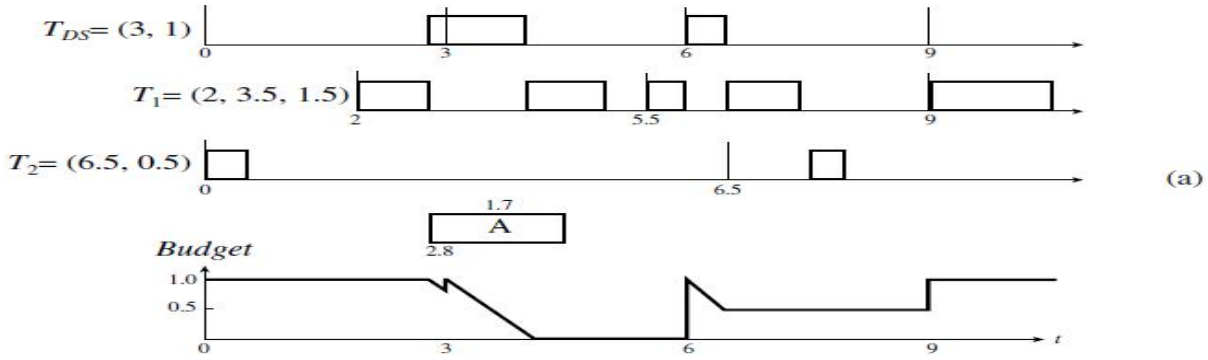


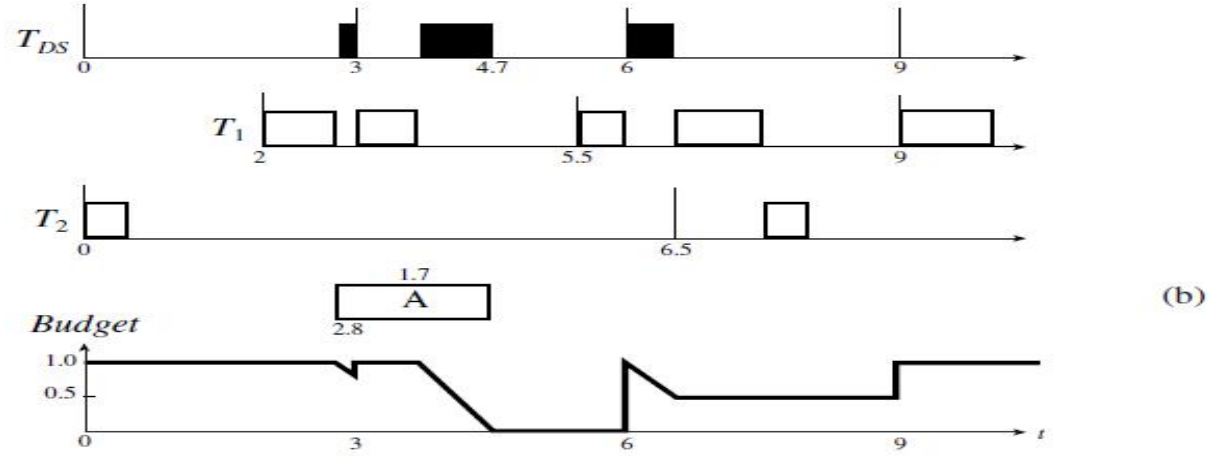
Figure (b) shows the same periodic tasks and the deferrable server scheduled according to the EDF algorithm. At any time, the deadline of the server is equal to the next replenishment time.

1.At time 2.8, the deadline of the deferrable server is 3.0. Consequently, the deferrable server executes at the highest-priority beginning at this time.

2. At time 3.0, when the budget of the deferrable server is replenished, its deadline for consuming this new unit of budget is 6. Since the deadline of *J*1,1 is sooner, this job has a higher priority. The deferrable server is preempted.

3. At time 3.7, *J*1,1 completes. The deferrable server executes until time 4.7 when its budget is exhausted.

4. At time 6 when the server's budget is replenished, its deadline is 9, which is the same as the deadline of the job *J*1,2. Hence, *J*1,2 would have the same priority as the server.



SPORADIC SERVERS

We have just seen that a deferrable server may delay lower-priority tasks for more time than a period task with the same period and execution time. This section describes a class of bandwidth-preserving servers, called *sporadic servers*,that are designed to improve over a deferrable server in this respect. The consumption and replenishment rules of sporadic server algorithms ensure that each sporadic server with period p_s and budget e_s never demands more processor time than the periodic task (p_s , e_s) in any time interval. Consequently, we can treat the sporadic server exactly like the periodic task (p_s , e_s) when we check for the schedulability of the system. A system of periodic tasks containing a

sporadic server may be schedulable while the same system containing a deferrable server with the same parameters is not.

Sporadic Server in Fixed-Priority Systems:

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

Since our focus here is on the consumption and replenishment rules of the server, we assume that we have chosen the parameters p_s and e_s and have validated that the periodic task (p_s , e_s) and the system **T** are schedulable according to the fixed-priority algorithm used by the system. When doing the schedulability test, we assume that the relative deadline for consuming the server budget is finite but arbitrary. In particular, we allow this deadline to belarger than p_s . During an interval when the aperiodic job queue is never empty, the server behaves like the periodic task (p_s , e_s) in which some jobs may take longer than one period to complete.

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

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

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

Simple Sporadic Server:

In its simpliest form, a sporadic server is governed by the following consumption and replenishment rules. We call such a server a simple sporadic server. A way to implement the server is to have the scheduler monitor the busy intervals of TH and maintain information on BEGIN and END.

- **Consumption Rules of Simple Fixed-Priority Sporadic Server:** At any time t after tr, the server's execution budget is consumed at the rate of 1 per unit time until the budget is exhausted when either one of the following two conditions is true. When these conditions are not true, the server holds its budget.

C1→ The server is executing.

C2→ The server has executed since tr and END < t.

• **Replenishment Rules of Simple Fixed-Priority Sporadic Server**:

R1 Initially when the system begins execution and each time when the budget is replenished, the execution budget = e_s, and t_r = the current time.

R2 At time t_f , if END = t_f , t_e = max(t_r, BEGIN). If END < t f , t_e = t_f .The next replenishment time is set at $t_e + p_s$.

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

(a) If the next replenishment time $t_e + p_s$ is earlier than t_f , the budget is replenished as soon as it is exhausted.

(b) If the system T becomes idle before the next replenishment time $t_e + p_s$ and becomes busy again at tb, the budget is replenished at min($t_e + p_s$, t_b).

Rules C1 and R1 are self-explanatory. Equivalently, rule C2 says that the server consumes its budget at any time t if it has executed since tr but at t, it is suspended and the higher-priority subsystem TH is idle. Rule R2 says that the next replenishment time is ps units after tr (i.e., the effective replenishment time te is tr) only if the higher-priority subsystem TH has been busy throughout the interval (t_r, t_f). Otherwise, te is later; it is the latest instant at which an equal or lower-priority task executes (or the system is idle) in (t_r, t_f).

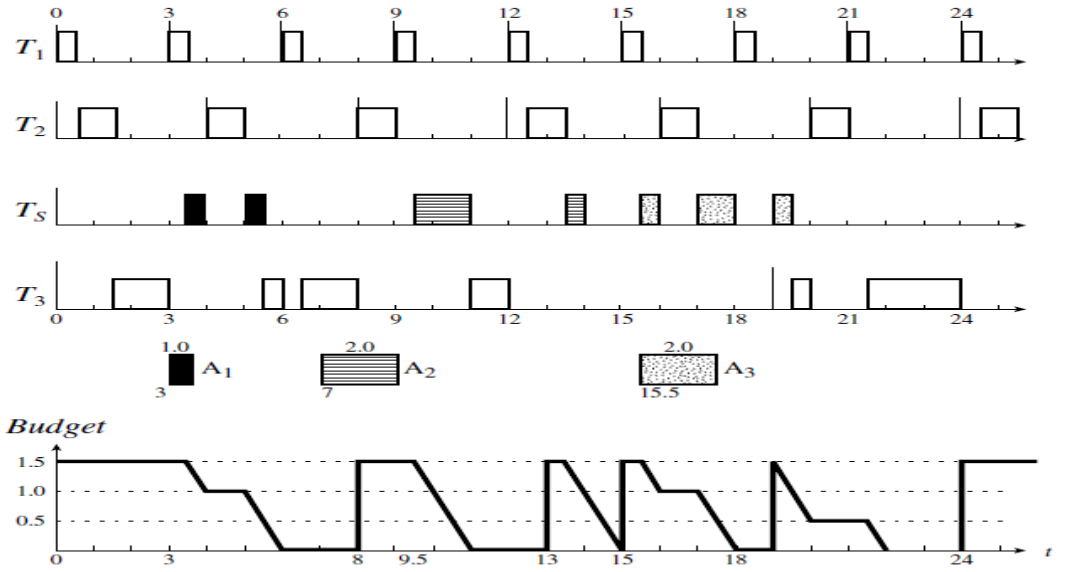


FIGURE 7–8 Example illustrating the operations of a simple sporadic server: $T_1 = (3, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$, $T_s = (5, 1.5)$.

Example:

Initially the budget of the server (5, 1.5) is 1.5. It is scheduled rate-monotonically with three periodic tasks: T1 = (3, 0.5), T2 = (4, 1.0), and T3 = (19, 4.5). They are schedulable even when the aperiodic job queue is busy all the time.

1. From time 0 to 3, the aperiodic job queue is empty and the server is suspended. Since it has not executed, its budget stays at 1.5. At time 3, the aperiodic job A1 with execution time 1.0 arrives; the server becomes ready. Since the higher-priority task (3, 0.5) has a job ready for execution, the server and the aperiodic job wait.

2. The server does not begin to execute until time 3.5. At the time, tr is 0, BEGIN is equal to 3, and END is equal to 3.5. According to rule R2, the effective replenishment time te is equal to max(0, 3.0) = 3, and the next replenishment time is set at 8.

3. The server executes until time 4; while it executes, its budget decreases with time.

4. At time 4, the server is preempted by T2. While it is preempted, it holds on to its budget.

5. After the server resumes execution at 5, its budget is consumed until exhaustion because first it executes (C1) and then, when it is suspended again, T1 and T2 are idle (or equivalently, END, which is 5.0, is less than the current time) (C2).

6. When the aperiodic job A2 arrives at time 7, the budget of the server is exhausted; the job waits in the queue.

7. At time 8, its budget replenished (R3), the server is ready for execution again.

8. At time 9.5, the server begins to execute for the first time since 8. te is equal to the latest replenishment time 8. Hence the next replenishment time is 13. The server executes until its budget is exhausted at 11; it is suspended and waits for the next replenishment time. In the meantime, A2 waits in the queue.

9. Its budget replenished at time 13, the server is again scheduled and begins to execute at time 13.5. This time, the next replenishment time is set at 18. However at 13.5, the periodic task system T becomes idle. Rather than 18, the budget is replenished at 15, when a new busy interval of T begins, according to rule R3b.

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

Enhancement of fixed- priority sporadic server:

The sporadic servers somewhat more complex and pay for slightly higher scheduling overhead, there are many ways to replenish the execution budget more aggressively and preserve it longer. One point to keep in mind in this discussion is the relationship between the replenishment and consumption rules. If we change one rule, we may need to change some other rule(s) in order for the entire set of rules to remain correct.

Sporadic/Background server:

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

• ***Consumption rules of simple sporadic/background servers*** are the same as the rules of simple sporadic servers except when the period task system is idle. As long as the periodic task system is idle, the execution budget of the server stays at e_s.

• ***Replenishment rules of simple sporadic/background servers*** are same as those of the simple sporadic server except R3b. The budget of a sporadic/background server is replenished at the beginning of each idle interval of the periodic task system. t_r is set at the end of the idle interval.

Cumulative Replenishment:

a simple server is not allowed to cumulate its execution budget from replenishment to replenishment. This simplifies the consumption and replenishment rules. A way to give the server more budget is to let the server keep any budget that remains unconsumed.

SpSL Sporadic Servers: a SpSL server with period p_s and execution budget e_s emulates several periodic tasks with the same period and total execution time equal to e_s.

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

We now describe three bandwidth preserving server algorithms that offer a simple way to schedule aperiodic jobs in deadline-driven systems. They are constant utilization, total bandwidth, and weighted fair-queueing algorithms. These algorithms belong to a class of algorithms that more or less emulate the Generalized Processor Sharing (GPS) algorithm. GPS, sometimes called fluid-flow processor sharing, is an idealized weighted round robin algorithm; it gives each backlogged server in each round an infinitesmally small time slice of length proportional to the server size.

In particular, each server maintained and scheduled according to any of these algorithms offers timing isolation to the task(s) executed by it; by this statement, we mean that the worst case response times of jobs in the task are independent the processor-time demands of tasks executed by other servers. While such a server works in a way that is similar to sporadic servers, its correctness relies on a different principle.

Constant Utilization Server Algorithm: The server is defined by its size, which is its instantaneous utilization u˜s; this fraction of processor time is reserved for the execution of aperiodic jobs. As with deferrable servers, the deadline d of a constant utilization server is always defined. It also has an execution budget which is replenished according to the replenishment rules described below. The server is eligible and ready for execution only when its budget is nonzero. When the server is ready, it is scheduled with the periodic tasks on the EDF basis. While a sporadic server emulates a periodic task, a constant utilization server emulates a sporadic task with a constant instantaneous utilization, and hence its name.

Consumption and Replenishment Rules:

The consumption rule of a constant utilization server, as well as that of a total bandwidth or weighted fair-queueing server, is quite simple. A server consumes its budget only when it executes. You will see shortly that such a server never has any budget when there is no aperiodic job ready for execution. Hence the problem of dealing with chunks of budget never arises. The budget of a basic constant utilization server is replenished and its deadline set according to the following rules. In the description of the rules, u˜s is the size of the server, es is its budget, and d is its deadline. t denotes the current time, and e denotes the execution time of the job at the head the aperiodic job queue. The job at the head of the queue is removed when it completes. The rules assume that the execution time e of each aperiodic job becomes known when the job arrives. We will return later to discuss how to remove this restriction.

Replenishment Rules of a Constant Utilization Server of Size u˜s

R1 Initially, es = 0, and d = 0.

R2 When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue,

(a) if $t < d$, do nothing;

(b) if $t \geq d$, $d = t + e/u˜s$, and $es = e$.

R3 At the deadline d of the server,

(a) if the server is backlogged, set the server deadline to $d + e/u˜s$ and $es = e$;

(b) if the server is idle, do nothing.

In short, a constant utilization server is always given enough budget to complete the job at the head of its queue each time its budget is replenished. Its deadline is set so that its instantaneous utilization is equal to u˜s.

Scheduling Aperiodic Jobs with Unknown Execution Times:

In the description of the constant utilization server algorithm, we assume that the execution times of aperiodic become known upon their arrival. This restrictive assumption can be removed by modifying the replenishment rules of constant utilization (or total bandwidth) servers. One way is to give the server a fixed size budget es and fixed period es/u˜s just like sporadic and deferrable servers. Since the execution time of each aperiodic job can be determined after it is completed with little overhead, we can adjust the server deadline based on this knowledge upon the completion of the job. Specifically, when an aperiodic job with execution time e shorter than es completes, we reduce the current deadline of the server by (es − e)/u˜s units before replenishing the next es units of budget and setting the deadline accordingly. This action clearly can improve the performance of the server and does not make the instantaneous utilization of the server larger than u˜s. An aperiodic job with execution time larger than es is executed in more than one server period. We can treat the last chunk of such a job in the manner described above if the execution time of this chunk is less than es

Total Bandwidth Server Algorithm:

To motivate the total bandwidth server algorithm [SpBu], Suppose that A3 were to arrive at time 14 instead. Since 14 is before the current server deadline 15, the scheduler must wait until time 15 to replenish the budget of the constant utilization server. A3 waits in the interval from 14 to 15, while the processor idles! Clearly, one way to improve the responsiveness of the server is to replenish its budget at time 14. This is exactly what the total bandwidth server algorithm does.

Specifically, the total bandwidth server algorithm improves the responsiveness of a constant utilization server by allowing the server to claim the background time not used by periodic tasks. This is done by having the scheduler replenish the server budget as soon as the budget is exhausted if the server is backlogged at the time or as soon as the server becomes backlogged. We now show that a constant utilization server works correctly if its budget is replenished in this aggressive manner.

In particular, we can change the replenishment rules as follows and get a total bandwidth server. . Replenishment Rules of a Total Bandwidth Server of size u˜s R1 Initially, es = 0 and d = 0. R2 When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue,

set d to max(d, t) + e/u˜s and es = e. R3

When the server completes the current aperiodic job, the job is removed from its queue.

(a) If the server is backlogged, the server deadline is set to d + e/u˜s, and es = e.

(b) If the server is idle, do nothing.

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

Scheduling Aperiodic and Sporadic Jobs in Priority-Driven Systems respectively, A3 would be completed at time 17.5 if it were executed by a total bandwidth server but would be completed at 19 by a constant bandwidth server. Clearly, a total bandwidth server does not behave like a sporadic task with a constant instantaneous utilization.

To see why it works correctly, let us examine how the server affects periodic jobs and other servers when its budget is set to e at a time t before the current server deadline d and its deadline is postponed to the new deadline d= d + e/u˜s. In particular, we compare the amount of processor time demanded by the server with the amount of time demanded by a constant utilization server of the same size u˜s before the deadline di,k of a periodic job Ji,k whose deadline is later than the server's new deadline d. (We do not need to consider periodic jobs with deadlines earlier than d because they have higher priorities than the server.) If the job Ji,k is ready at t, then the amounts of time consumed by both servers are the same in the interval from t to di,k . If Ji,k is not yet released at t, then the time demanded by the total bandwidth server in the interval (ri,k , d] is less than the time demanded by the constant utilization server because the total bandwidth server may have executed before ri,k and has less budget in this interval. In any case, the total bandwidth server will not cause a job such as Ji,k to miss its deadline if the constant utilization server will not. By a similar argument, we can show that a total bandwidth server will not cause another server to become unschedulable if a constant utilization server of the same size will not. We state the correctness of constant utilization and total bandwidth server algorithms in the following corollaries so we can refer to them later. In the statement of the corollaries, we use the expression "a server meets its deadline" .

By this, we mean that the budget of the server is always consumed by the deadline set at the time when the budget was replenished. If we think of the server as a sporadic task and each replenishment of the budget of e units as the release of a sporadic job that has this execution time and the corresponding deadline, then every job in this sporadic task completes in time.

Fairness and Starvation:

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

It is well known in communication literature that the virtual clock algorithm (i.e., non preemptive total bandwidth server algorithm) is unfair. To illustrate that this is also true for the total bandwidth server algorithm, let us consider a system consisting solely of two total bandwidth servers, *TB*1 and *TB*2, each of size 0.5. Each server executes an aperiodic task; jobs in the task are queued in the server's own queue. It is easy to see that if both servers are never
idle, during any time interval of length large compared to the execution times of their jobs,the total amount of time each server executes is approximately equal to half the length of the interval. In other words, each server executes for its allocated fraction of time approximately**.**

Definition of Fairness:

Since fairness is not an important issue for periodic tasks, we confine our attention here to systems containing only aperiodic and sporadic jobs. Specifically, we consider a system consisting solely of n (> 1) servers. Each server executes an aperiodic or sporadic task. For $i = 1, 2, \ldots, n$, the size of the ith server is $\tilde{u}i$. n i=1 $\tilde{u}i$ is no greater than 1, and hence every server is schedulable. $\tilde{w}i(t1, t2)$, for $0 < t1 < t2$, denotes the total attained processor time of the ith server in the time interval $(t1, t2)$, that is, the server executes for $\tilde{w}i(t1, t2)$ units of time during this interval.

The ratio $\tilde{}wi\ (t1, t2)/\tilde{}\ ui$ is called the *normalized service* attained by the *i*th server [StVa98a]. A scheduler (or the scheduling algorithm used by the scheduler) is *fair* in the interval $(t1, t2)$ if the normalized services attained by all servers that are backlogged during the interval differ by no more than the *fairness threshold* $FR \geq 0$. In the ideal case, FR is equal to zero, and

$\tilde{}wi\ (t1, t2)\tilde{}w\ j\ (t1, t2) = \tilde{}\ ui\tilde{}\ u\ j$ for any $t2 > t1$ and *i*th and *j* th servers that are backlogged throughout the time interval $(t1, t2)$. Equivalently, $\tilde{}w\ i\ (t1, t2) = \tilde{}ui\ (t2 - t1)$.

Preemptive Weighted Fair-Queuing Algorithm:

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

The WFQ algorithm is designed to ensure fairness [DeKS] among multiple servers. The algorithm closely resembles the total bandwidth server algorithm. Both are greedy, that is, work conserving. Both provide the same schedulability guarantee, and hence, the same worst-case response time. At a quick glance, the replenishment rules of a WFQ server appear to be the same as those of a total bandwidth server, except for how the deadline is computed at each replenishment time. This difference, however, leads to a significant difference in their behavior: The total bandwidth server algorithm is unfair, but the WFQ algorithm gives bounded fairness.

Rules of Preemptive Weighted Fair-Queueing Algorithm:

We are now ready to state the rules that define the WFQ 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.
- $f\ ni$ denotes the finish number of the server FQi , ei its budget, and $\tilde{}\ ui$ its size. e denotes the execution time of the job at the head of the server queue.
- Ub 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 Ub were updated.

Initialization Rules

I1 For as long as all servers (and hence the system) are idle, $F\ N = 0$, $Ub = 0$, 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 FQk and starts a busy interval of the system,

(a) $t-1 = t$, and increment Ub by $\tilde{} uk$, and

(b) set the budget ek of FQk to e and its finish number $f nk$ to $e/\tilde{} uk$.

Rules for Updating F N and Replenishing Budget of FQi during a System Busy Interval

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

(a) increment system finish number $F N$ by $(t - t-1)/Ub$,

(b) $t-1 = t$, and increment Ub by $\tilde{} ui$, and

(c) set budget ei of FQi to e and its finish number $f ni$ to $F N + e/\tilde{} ui$ and place theserver in the ready server queue in order of nonincreasing finish numbers.

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

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

(b) if the server becomes idle, update Ub and $F N$ as follows:

i. Increment the system finish number $F N$ by $(t - t-1)/Ub$,

ii. $t-1 = t$ and decrement Ub by $\tilde{} ui$.

SLACK STEALING IN DEADLINE-DRIVEN SYSTEMS:

We now describe how to do slack-stealing in priority-driven systems. we first focus on systems where periodic tasks are scheduled according to the EDF algorithm.

It is convenient for us to think 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.

In principle slack stealing in a priority-driven system is almost as straightforward as in a clock-driven system. The key step in slack stealing is the computation to determine whether the system has any slack. While this is a simple step in a clock-driven system, it is considerably more complex in a priority-driven system.

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.

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.

Optimality Criterion and Design Consideration:

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. (Using a similar argument, Tia [Tia] also showed that *no on-line slack-stealing algorithm can minimize the average response time of all the aperiodic jobs*; however a clairvoyant algorithm can.)

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.

SCHEDULING OF SPORADIC JOBS:

The scheduler performs an acceptance test on each sporadic job upon its arrival. We now describe how to do acceptance tests in priority-driven systems. 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.

A Simple Acceptance Test in Deadline-Driven Systems:

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 $1 - \_$. If the scheduler accepts the job, the (absolute) deadline d of S divides the time after t into two disjoint time intervals: the interval $I1$ at and before d and the interval $I2$ after d. The job S is active in the former but not in the latter. Consequently, the total densities $\_s,1$ and $\_s,2$ 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 *ns* 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 *ns* + 1 disjoint intervals: *I*1, *I*2, . . . , *Ins*+1. *I*1 begins at *t* and ends at the first (i.e., the earliest) deadline in the list. For $1 \leq k \leq ns$, each subsequent interval *Ik*+1 begins when the previous interval *Ik* ends and ends at the next deadline in the list or, in the case of *Ins*+1, at infinity. (Some of these intervals have zero length when the sporadic jobs have nondistinct deadlines.) The scheduler also keeps up-to-date the total density _*s,k* of the sporadic jobs that are active during each of these intervals. Let *Il* 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 $e/(d - t) + \blacktriangledown s,k \leq 1 - \blacktriangledown$ for all $k = 1, 2, \ldots l$.

If these conditions are satisfied and *S* is accepted, the scheduler divides the interval *Il* into two intervals: The first half of *Il* ends at *d*, and the second half of *Il* begins immediately after *d*.We now call the second half *Il*+1 and rename the subsequent intervals *Il*+2, . . . , *Ins*+2. (Here, we rename the intervals for the sake of clarity. In the actual implementation of the acceptance test, this step is not necessary provided some appropriate data structure is used to allow efficient insertion and deletion of the intervals and update of the total densities associated with individual intervals.) The scheduler increments the total density _*s,k* of all active sporadic jobs in each of the intervals *I*1, *I*2, . . . , *Il* 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(Ns)* where *Ns* is the maximum number of sporadic jobs that can possibly be in the system at the same time. 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.

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

The time complexity of the acceptance test based on this algorithm is *O(n + Ns)*. Other existing algorithms for an acceptance test based on static-slack computation in deadline-driven systems. These algorithms have pseudo polynomial complexity *O(N + Ns)*. Alternatives are to compute the available slack dynamically or to subject periodic jobs to an acceptance test as well as sporadic jobs. Both alternatives have their own shortcomings. Except when an extreme high run-time overhead can be tolerated, we are forced to use dynamically computed lower bounds on the available slack. These bounds tend to be loose. As a consequence, the acceptance test is no longer optimal and may not have significant advantage over the simple acceptance test described earlier. Subjecting periodic jobs to an acceptable test is not only time consuming, especially since the acceptance test is pseudo polynomial time in complexity, but may also result in the rejection of some periodic jobs, which is incorrect according to the correctness criterion used here.

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 *Jl* . In addition to the slack of *Jl* , which can be used to execute *S*1, the interval *(dl , d*] is also available. Hence the slack of *S*1 is given by $\sigma s,1(t) = \sigma l\,(t) + (d - dl\,) - e$

The slack of the job Jl 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, σl (t) = ω(l; l) − I −_ dk>d ξk.

When the deadline of the first periodic job in the hyperperiod is later than *d*, *σl (t)* and *dl* are zero by definition, and *σs,*1*(t)* is simply equal to *d* − *e*. In the first step of the acceptance test, the scheduler computes *σs,*1*(t)* in this way. It rejects the new sporadic job *S*1 if *σs,*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 *σk(t)* of all the periodic jobs *Jk* for *k* = *l* + 1, *l* + 2, . . . , *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 *Si (t, d, e)* is tested, there may be *ns* sporadic jobs in the system waiting to complete. Similar to Eq. (7.12), the slack of *Si* at time *t* can be expressed in terms of the slack *σl (t)* of its leverage job *Jl* as follows.

σs,i (t) = σl (t) + (d − dl) − e − T E −_ ds,k≤ des,k −_ ds ,k>d ξs,k

Acceptance Tests for Sporadic Jobs with Arbitrary Deadlines.

Finally, let us consider a sporadic job *Si (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* ≤ *(z* − 1*)H* < *d* ≤ *zH*. For this sporadic job, the leverage job *Jl* 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 non existing job whose slack is 0 and deadline is *(z* − 1*)H*, if there is no such periodic job.

One way to schedule sporadic jobs in a fixed-priority system is to use a sporadic server to

execute them. Because the server *(ps , es)* has *es* units of processor time every *ps* 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.

As always, we assume that accepted sporadic jobs are ordered among themselves on an EDF basis. When the first sporadic job *S*1*(t, ds,*1*, es,*1*)* arrives, the server has at least

*_(ds,*1 − *t)/ps_es* units of processor time before the deadline of the job. Therefore, the scheduler accepts *S*1 if the slack of the job *σs,*1*(t) = _(ds,*1 − *t)/ps_es* − *es,*1 is larger than or equal to 0.

To decide whether a new job *Si (t, ds,i , es,i)* is acceptable when there are *ns* sporadic

jobs in the system, the scheduler computes the slack *σs,i* of *Si* according to *σs,i (t) = _(ds,i* − *t)/ps_es* − *es,i* − *_ ds,k<ds,I (es,k* − *ξs,k)* where *ξs,k* is the execution time of the completed portion of the existing sporadic job *Sk*. The new job *Si* cannot be accepted if its slack is less than 0. If *σs,i (t)* is no less than 0, the scheduler then checks whether any existing sporadic job *Sk* whose deadline is after *ds,i* may be adversely affected by the acceptance of *Si* . This can easily be done by checking whether

the slack $\sigma s,k(t)$ of Sk at the time is equal to or larger than es,i . The scheduler accepts Si if $\sigma s,k\ (t) - es,i \geq 0$ for every existing sporadic job Sk with deadline equal to or later than ds,i . If the scheduler accepts Si , 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 ds,i by the execution time es,i of the new job.

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, 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 while they may be acceptable if no slack is used. The presence of sporadic jobs further increases the complexity of slack computation. In contrast, both the implementation and validation of the system are straightforward when we use bandwidth preserving servers to execute aperiodic or sporadic jobs.

A TWO-LEVEL SCHEME FOR INTEGRATED SCHEDULING:

By design, the two-level scheme allows different applications to use different scheduling . Hence, each application can be scheduled in a way best for the application. More importantly, the schedulability and real-time performance of each application can be determined independently of other applications executed on the same processor. By emulating an infinitesmally fine-grain time slicing scheme, the two-level scheme creates a slower virtual processor for each applications in the system.

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 required capability and its server is schedulable. In contrast, a nonpredictable application requires a server of a size larger than than its required capability, as we will explain below.

Nonpreemptively Scheduled Applications:

Specifically, according to the two-level scheme, each nonpreemptively scheduled application **T***i* is executed by a constant utilization server whose size $\tilde{} ui$ is equal to the required capacity si of the application. The server scheduler orders jobs in **T***i* according to the algorithm used by **T***i*. Let B denote the maximum execution time of nonpreemptable sections3 of all jobs in all applications in the system and Dmin denote the minimum of relative deadlines of all jobs in these applications. All servers are schedulable if the total size of all servers is no greater than $1 - B/D$min. To show that every job in a nonpreemptively scheduled real-time application **T***i* completes in time when scheduled according to the two-level scheme, we consider a time instant at which the OS scheduler replenishes the server budget and sets the server deadline to execute a job in **T***i* . This time instant is the same as the scheduling decision time at which the job would be scheduled if **T***i* were executed alone on a slow processor whose speed is si times the speed of the physical processor. The job would complete on the slow processor at the deadline of the server. Since the server is schedulable, it surely will execute for as long as the execution time of the job (hence complete the job) by the server deadline according to the two-level scheme.4

Applications Scheduled according to a Cyclic Schedule

Each application **T***i* that is scheduled according to a cyclic schedule is also executed by a constant utilization server of size equal to *si* . The OS replenishes the budget of the server at the beginning of each frame. At each replenishment, the budget is set to *si 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 pre computed cyclic schedule.

Preemptively Scheduled Predictable Applications

As with other predictable applications, the size ˜ *ui* of the server for a predictable application **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.

Scheduling Nonpredictable Applications

In general, the actual release-times of some jobs in a non predictable application **T***i* may be unknown. It is impossible for its server scheduler to determine the next release-time *t*_ of **T***i* precisely. Therefore, some priority inversion due to over replenishment of the server budget is unavoidable. Fortunately, the bad effect on the schedulability of **T***i* can be compensated by making the size ˜ *ui* 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 **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 **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 **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*(er, (t* _*e*− *t)* ˜ *ui)* and the server deadline at min*(t* + *er* / ˜ *ui* , *t* _*e). ε* is a design parameter. It is called the *quantum size* of the two-level scheduler. In the absence of any knowledge on the release-times of jobs in an application, the OS scheduler replenishes the budget of its server every ε units of time. Hence, the smaller the quantum size, the more frequent the server replenishments, and the higher the server maintenance overhead. However, the size of the server required to execute the application grows with the quantum size.

Required Server Size. It is easy to see that if **T***i* were to execute alone on a slower processor, it would not have any context switch from the current replenishment time to ε time units before the new server deadline. The extra budget that the OS schedulermay inadvertently give to the server is no greater than ε ˜ *ui* . Hence, the length of priority inversion due to the overreplenishment of the server budget is at most ε ˜ *ui* . We can treat this time as the blocking time of any higher-priority job that may be released between the current replenishment time and server deadline. Let *Di,*min be the minimum relative deadline of all jobs in **T***i* . Section 6.8 tells us that if the size of the server for **T***i* satisfies the inequality $si+\varepsilon$ ˜ *ui /Di,*min ≤ ˜*ui* , 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

˜ *ui* = *si Di,*min *Di,*min − ε

In conclusion, a nonpredictable application **T***i* with required capacity *si* and minimum relative
deadline *Di,*min is schedulable according to the two-level scheme if the size of its server is given by the expression above and the total size of all servers in the system is no greater than 1 − *B/D*min.