Technical Report

CMU/SEI-89-TR-11
ESD-TR-89-19

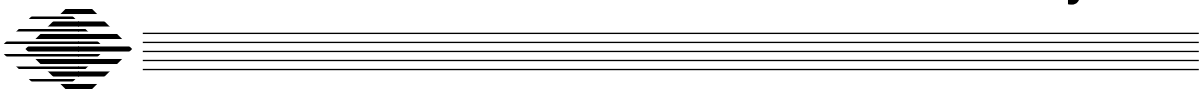
Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System

Brinkley Sprunt
Lui Sha
John Lehoczky

April 1989

Technical Report
CMU/SEI-89-TR-11
ESD-TR-89-19
April 1989

Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System



Brinkley Sprunt

Department of Electrical
and Computer Engineering

Lui Sha

Real-Time Scheduling in Ada Project

John Lehoczky

Department of Statistics

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Asset Source for Software Engineering Technology (ASSET) / 1350 Earl L. Core Road ; P.O. Box 3305 / Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / Fax: (304) 284-9001 / e-mail: sei@asset.com / WWW: http://www.asset.com/sei.html

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Reviewed and edited by Information Management, a function of the Technology Transition Program, Software Engineering Institute.]

Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System

Abstract: A real-time system consists of both aperiodic and periodic tasks. Periodic tasks have regular arrival times and hard deadlines. Aperiodic tasks have irregular arrival times and either soft or hard deadlines. In this paper, we present a new algorithm, the Sporadic Server algorithm, that greatly improves response times for soft-deadline aperiodic tasks and can guarantee hard deadlines for both periodic and aperiodic tasks. The operation of the Sporadic Server algorithm, its performance, and schedulability analysis are discussed and compared with previous, published aperiodic service algorithms.

1. Introduction: The Real-Time Scheduling Problem

Real-time systems are used to control physical processes that range in complexity from automobile ignition systems to controllers for flight systems and nuclear power plants. In these systems, the correctness of system functions depends upon not only the results of computation but also the times at which results are produced.

A real-time task is generally placed into one of four categories based upon its arrival pattern and its deadline. If meeting a given task's deadline is critical to the system's operation, then the task's deadline is considered to be *hard*. If it is desirable to meet a task's deadline but occasionally missing the deadline can be tolerated, then the deadline is considered to be *soft*. Tasks with regular arrival times are called *periodic* tasks. A common use of periodic tasks is to process sensor data and update the current state of the real-time system on a regular basis. Periodic tasks, typically used in control and signal processing applications, have hard deadlines. Tasks with irregular arrival times are *aperiodic* tasks. Aperiodic tasks are used to handle the processing requirements of random events such as operator requests. An aperiodic task typically has a soft deadline. Aperiodic tasks that have hard deadlines are called *sporadic* tasks. We assume that each task has a known worst case execution time. In summary, we have:

- **Hard and soft deadline periodic tasks.** A periodic task has a regular interarrival time equal to its period and a deadline that coincides with the end of its current period. Periodic tasks usually have hard deadlines, but in some applications the deadlines can be soft.

- **Soft deadline aperiodic tasks.** An aperiodic task is a stream of jobs arriving at irregular intervals. Soft deadline aperiodic tasks typically require a fast average response time.

- **Sporadic tasks.** A sporadic task is an aperiodic task with a hard deadline and a *minimum* interarrival time [6]. Note that without a minimum interarrival time restriction, it is impossible to guarantee that a deadline of a sporadic task would always be met.

To meet the timing constraints of the system, a scheduler must coordinate the use of all system resources using a set of well understood real-time scheduling algorithms that meet the following objectives:

- Guarantee that tasks with hard timing constraints will always meet their deadlines.

- Attain a high degree of schedulable utilization for hard deadline tasks (periodic and sporadic tasks). Schedulable utilization is the degree of resource utilization at or below which all hard deadlines can be guaranteed. The schedulable utilization attainable by an algorithm is a measure of the algorithm's utility: the higher the schedulable utilization, the more applicable the algorithm is for a range of real-time systems.

- Provide fast average response times for tasks with soft deadlines (aperiodic tasks).

- Ensure scheduling stability under transient overload. In some applications, such as radar tracking, an overload situation can develop in which the computation requirements of the system exceed the schedulable resource utilization. A scheduler is said to be stable if during overload it can guarantee the deadlines of critical tasks even though it is impossible to meet all task deadlines.

The quality of a scheduling algorithm for real-time systems is judged by how well the algorithm meets these objectives.

This paper develops advanced algorithms to schedule aperiodic tasks. For soft deadline aperiodic tasks, the goal is to provide fast average response times. For hard deadline aperiodic tasks (sporadic tasks), the goal is to guarantee that their deadlines will always be met. The new algorithms presented herein meet both of these goals and are still able to guarantee the deadlines of hard deadline periodic tasks. For simplicity, we assume that periodic tasks have hard deadlines and constant execution times.[1] In Chapter 2 we will review related work on scheduling periodic and aperiodic tasks. Chapter 3 introduces the *Sporadic Server* (SS) algorithm for scheduling aperiodic tasks and illustrates its operation with several examples. This section also addresses the schedulability issues of the SS algorithm, the interaction of the SS algorithm and priority inheritance protocols, and compares the performance of the SS algorithm with previous algorithms. Chapter 4 addresses the scheduling of sporadic tasks and discusses the use of a *deadline monotonic* priority assignment for scheduling sporadic tasks that have short deadlines and long interarrival times. Finally, Chapter 5 presents a summary and conclusions.

[1]Readers who are interested in the subject of taking advantage of the stochastic execution time of periodic tasks are referred to [10].

2. Background and Related Work

2.1. Scheduling Periodic Tasks

A well understood scheduling algorithm for guaranteeing the hard deadlines of periodic tasks in a multiprogrammed real-time system is Liu and Layland's rate monotonic scheduling algorithm [5]. Under this algorithm, fixed priorities are assigned to tasks based upon the rate of their requests (i.e., a task with a relatively short period is given a relatively high priority). Liu and Layland proved that this algorithm is the optimum static preemptive scheduling algorithm for periodic tasks with hard deadlines. The algorithm guarantees that **n** periodic tasks can always be guaranteed to meet their deadlines if the resource utilization of the tasks is less than $\mathbf{n}(2^{1/\mathbf{n}}-1)$, which converges to ln 2 (= 69%) for large **n**. The rate monotonic algorithm can be used as a basis to develop a family of scheduling algorithms that address a wide range of practical problems. The rate monotonic algorithm has the following favorable qualities for scheduling real-time systems:

- **High schedulable utilization.** Although Liu and Layland show a low scheduling bound for the rate monotonic algorithm, this bound is pessimistic and represents the absolute worst case conditions. Lehoczky, Sha, and Ding [2] performed an exact characterization and stochastic analysis for a randomly generated set of periodic tasks scheduled by the rate monotonic algorithm and found that the average scheduling bound is usually much better than the worst case. They concluded that a good approximation to the threshold of schedulability for the rate monotonic algorithm is 88%. In fact, with the period transformation method, the utilization threshold can, in principle, be arbitrarily close to 100% [9]. As an example of the high degree of schedulable utilization attainable with the rate monotonic algorithm, a schedulable utilization level of 99% was achieved for the Navy's Inertial Navigation System [1].

- **Stability under transient overload.** Another concern for scheduling algorithms is transient overload, the case where stochastic execution times can lead to a desired utilization greater than the schedulable utilization bound of the task set. To handle transient overloads, Sha, Lehoczky, and Rajkumar describe a period transformation method for the rate monotonic algorithm that can guarantee that the deadlines of critical tasks can be met [7].

- **Aperiodic tasks.** A real-time system often has both periodic and aperiodic tasks. Lehoczky, Sha and Strosnider developed the *Deferrable Server* algorithm [3], which is compatible with the rate monotonic scheduling algorithm and provides a greatly improved average response time for soft deadline aperiodic tasks over polling or background service algorithms while still guaranteeing the deadlines of periodic tasks.

- **Resource sharing.** Although determining the schedulability of a set of periodic tasks that use semaphores to enforce mutual exclusion has been shown to be NP-hard [6], Sha, Rajkumar, and Lehoczky [8] have developed a priority inheritance protocol and derived a set of sufficient conditions under which a set of

periodic tasks that share resources using this protocol can be scheduled using the rate monotonic algorithm.

- **Low scheduling overhead.** Since the rate monotonic algorithm assigns a static priority to each periodic task, the selection of which task to run is a simple function. Scheduling algorithms that dynamically assign priorities, may incur a larger overhead because task priorities have to be adjusted in addition to selecting the highest priority task to execute.

As such, we will use the rate monotonic algorithm as the basis for scheduling soft deadline aperiodic tasks and sporadic tasks. In the next section, we will review the related work on aperiodic scheduling.

2.2. Scheduling Aperiodic Tasks

The scheduling problem for aperiodic tasks is very different from the scheduling problem for periodic tasks. Scheduling algorithms for aperiodic tasks must be able to guarantee the deadlines for hard deadline aperiodic tasks and provide good average response times for soft deadline aperiodic tasks even though the occurrence of the aperiodic requests are non-deterministic. The aperiodic scheduling algorithm must also accomplish these goals without compromising the hard deadlines of the periodic tasks.

Two common approaches for servicing soft deadline aperiodic requests are background processing and polling tasks. Background servicing of aperiodic requests occurs whenever the processor is idle (i.e., not executing any periodic tasks and no periodic tasks are pending). If the load of the periodic task set is high, then utilization left for background service is low, and background service opportunities are relatively infrequent. Polling consists of creating a periodic task for servicing aperiodic requests. At regular intervals, the polling task is started and services any pending aperiodic requests. However, if no aperiodic requests are pending, the polling task suspends itself until its next period, and the time originally allocated for aperiodic service is not preserved for aperiodic execution but is instead used by periodic tasks. Note that if an aperiodic request occurs just after the polling task has suspended, the aperiodic request must wait until the beginning of the next polling task period or until background processing resumes before being serviced. Even though polling tasks and background processing can provide time for servicing aperiodic requests, they have the drawback that the average wait and response times for these algorithms can be long, especially for background processing.

Figures 2-2 and 2-3 illustrate the operation of background and polling aperiodic service using the periodic task set presented in Figure 2-1. The rate monotonic algorithm is used to assign priorities to the periodic tasks yielding a higher priority for task **A**. In each of these examples, periodic tasks **A** and **B** both become ready to execute at time = 0. Figures 2-2 and 2-3 show the task execution from time = 0 until time = 20. In each of these examples two aperiodic requests occur: the first at time = 5 and the second at time = 12.

Figure 2-1: Periodic Task Set for Figures 2-2, 2-3, 2-4, and 2-5

The response time performance of background service for the aperiodic requests shown in Figure 2-2 is poor. Since background service only occurs when the resource is idle, aperiodic service cannot begin until time = 16. The response time of the two aperiodic requests are 12 and 6 time units respectively, even though both requests each need only 1 unit of time to complete.

Figure 2-2: Background Aperiodic Service Example

The response time performance of polling service for the aperiodic requests shown in Figure 2-3 is better than background service for the first request but not for the second. For this example, a polling server is created with an execution time of 1 time unit and a period of 5 time units which, using the rate monotonic algorithm, makes the polling server the highest priority task. The polling server's first period begins at time = 0. The lower part of Figure 2-3 shows the capacity (available execution time) of the polling server as a function of time. As can be seen from the upward arrow at time = 0 on the capacity graph, the execution time of the polling server is discarded during its first period because no aperiodic requests are pending. The beginning of the second polling period coincides with the first aperiodic request and, as such, the aperiodic request receives immediate service. However, the second

aperiodic request misses the third polling period (time = 10) and must wait until the fourth polling period (time = 15) before being serviced. Also note, that since the second aperiodic request only needs half of the polling server's capacity, the remaining half is discarded because no other aperiodic tasks are pending. Thus, this example demonstrates how polling can provide an improvement in aperiodic response time performance over background service but is not always able to provide immediate service for aperiodic requests.

Figure 2-3: Polling Aperiodic Service Example

The *Priority Exchange* (PE) and *Deferrable Server* (DS) algorithms, introduced by Lehoczky, Sha, and Strosnider in [3], overcome the drawbacks associated with polling and background servicing of aperiodic requests. As with polling, the PE and DS algorithms create a periodic task (usually of a high priority) for servicing aperiodic requests. However, unlike polling, these algorithms will preserve the execution time allocated for aperiodic service if, upon the invocation of the server task, no aperiodic requests are pending. These algorithms can yield improved average response times for aperiodic requests because of their ability to provide immediate service for aperiodic tasks. In particular, the DS algorithm has been shown capable of providing an order of magnitude improvement in the responsiveness of asynchronous class messages for real-time token rings [11]. These algorithms are called *bandwidth preserving* algorithms because they provide a mechanism for preserving the resource bandwidth allocated for aperiodic service if, upon becoming available, the bandwidth is not immediately needed. The PE and DS algorithms differ in the manner in which they preserve their high priority execution time.

The DS algorithm maintains its aperiodic execution time for the duration of the server's

period. Thus, aperiodic requests can be serviced at the server's high priority at anytime as long as the server's execution time for the current period has not been exhausted. At the beginning of the period of the DS, the server's high priority execution time is replenished to its full capacity.

The DS algorithm's method of bandwidth preservation is illustrated in Figure 2-4 using the periodic task set of Figure 2-1. For this example, a high priority server is created with an execution time of 0.8 time units and a period of 5 time units. At time = 0, the server's execution time is brought to its full capacity. This capacity is preserved until the first aperiodic request occurs at time = 5, at which point it is immediately serviced, exhausting the server's capacity by time = 6. At time = 10, the beginning of the third server period, the server's execution time is brought to its full capacity. At time = 12, the second aperiodic request occurs and is immediately serviced. Notice that although the second aperiodic request only consumes half the server's execution time, the remaining capacity is preserved, not discarded as in the polling example. Thus, the DS algorithm can provide better aperiodic responsiveness than polling because it preserves its execution time until it is needed by an aperiodic task.

Figure 2-4: Deferrable Server Example

Unlike the DS algorithm, the PE algorithm preserves its high priority execution time by exchanging it for the execution time of a lower priority periodic task. At the beginning of the PE server's period, the server's high priority execution time is replenished to its full capacity. If the highest priority execution time available is aperiodic time (as is the case at the beginning of the PE server's period) and aperiodic tasks are pending, then the aperiodic tasks are

serviced. Otherwise, the highest priority pending periodic task is chosen for execution and a priority exchange occurs. The priority exchange converts the high priority aperiodic time to aperiodic time at the assigned priority level of the periodic task. When a priority exchange occurs, the periodic task executes at the priority level of the higher priority aperiodic time, and aperiodic time is accumulated at the priority level of the periodic task. Thus, the periodic task advances its execution time, and the aperiodic time is not lost but preserved, albeit at a lower priority. Priority exchanges will continue until either the high priority aperiodic time is exhausted or an aperiodic request occurs in which case the aperiodic time is used to service the aperiodic request. Note that this exchanging of high priority aperiodic time for low priority periodic time continues until either the aperiodic time is used for aperiodic service or until the aperiodic time is degraded to the priority level of background processing (this complete degradation will occur only when no aperiodic requests arrive early enough to use the aperiodic time). Also, since the objective of the PE algorithm is to provide a low average response time for aperiodic requests, aperiodic requests win all priority ties. At all times the PE algorithm uses the highest priority execution time available to service either periodic or aperiodic tasks.

The PE algorithm's method of bandwidth preservation is demonstrated in Figure 2-5 using the periodic task set of Figure 2-1. In this example, a high priority PE server is created with an execution time of 1 time unit and a period of 5 time units. Since the PE algorithm must manage aperiodic time across all priority levels, the capacity of the PE server as a function of time consists of three graphs: one for each priority level. The PE server's priority is priority level 1, which corresponds to the highest priority level, followed by priority 2 for periodic task **A** and priority 3 for periodic task **B**. At time = 0, the PE server is brought to its full capacity but no aperiodic tasks are pending and a priority exchange occurs between priorities 1 and 2. The PE server gains aperiodic time at priority 2, and periodic task **A** executes at priority 1. At time = 4, task **A** completes and task **B** begins. Since no aperiodic tasks are pending, another exchange takes place between priority 2 and priority 3. At time = 5, the server's execution time at priority 1 is brought to its full capacity and is used to provide immediate service for the first aperiodic request. At time = 10, the server's priority 1 execution time is brought to full capacity and is then exchanged down to priority 2. At time = 12, the server's execution time at priority 2 is used to provide immediate service for the second aperiodic request. At time = 14.5, the remaining priority 2 execution time is exchanged down to priority 3. At time = 15, the newly replenished server time at priority 1 is exchanged down to priority 3. Finally, at time = 17.5, the remaining PE server execution time at priority 3 is discarded because no tasks, periodic or aperiodic, are pending. Thus, the PE algorithm can also provide improved response times for aperiodic tasks compared to the polling algorithm.

The PE and DS algorithms differ in their complexity and in their effect upon the schedulability bound for periodic tasks. The DS algorithm is a much simpler algorithm to implement than the PE algorithm, because the DS algorithm always maintains its high priority execution time at its original priority level and never exchanges its execution time with lower priority levels as does the PE algorithm. However, the DS algorithm does pay schedulability penalty (in terms of a lower schedulable utilization bound) for its simplicity.

Figure 2-5: Priority Exchange Server Example

Both algorithms require that a certain resource utilization be reserved for high priority aperiodic service. We refer to this utilization as the server size, $\mathbf{U_s}$, which is the ratio of the server's execution time to the server's period. The server size and type (i.e., PE or DS) determine the scheduling bound for the periodic tasks, $\mathbf{U_p}$, which is the highest periodic utilization for which the rate monotonic algorithm can always schedule the periodic tasks. Below are the equations developed in [3] for $\mathbf{U_p}$ in terms of $\mathbf{U_s}$ as the number of periodic tasks approaches infinity for the PE and DS algorithms:

$$\textbf{PE:} \quad \mathbf{U_p} = \ln\frac{2}{\mathbf{U_s}+1} \qquad\qquad \textbf{DS:} \quad \mathbf{U_p} = \ln\frac{\mathbf{U_s}+2}{2\mathbf{U_s}+1}$$

These equations show that for a given server size, $\mathbf{U_s}$ ($0 < \mathbf{U_s} < 1$), the periodic schedulability bound, $\mathbf{U_p}$, for the DS algorithm is lower than it is for the PE algorithm. These equations also imply that for a given periodic load, the server size, $\mathbf{U_s}$, for the DS algorithm is smaller than that for the PE algorithm. For example, with $\mathbf{U_p}$ = 60%, the server size for the PE algorithm is 10% compared to a server size for the DS algorithm of 7%.

The focus of this paper is to develop a general algorithm for scheduling both soft and hard deadline aperiodic tasks. Such an algorithm should provide a responsiveness comparable to that attainable with the PE and DS algorithms for soft deadline aperiodic tasks. For hard deadline aperiodic tasks, the algorithm should provide a general technique for allocating resource utilization to guarantee that the deadline will always be met.

By comparing the PE and DS algorithms, we can identify the relative merits of each. The advantage of the DS algorithm over the PE algorithm is that it is conceptually much simpler, and thus easier to implement. The PE algorithm must manage aperiodic time across all priority levels in the system, whereas the DS algorithm maintains its execution time at its original priority level. The simple bandwidth preservation technique of the DS algorithm also implies that, on average, the priority of the server's execution time will be higher than it would be for an equivalently sized PE server because the DS algorithm preserves its high priority execution time at its original priority level for the duration of its period. In contrast, the PE algorithm must either use its high priority execution time for aperiodic service or trade it for lower priority periodic time. However, the PE algorithm has a server size advantage over the DS algorithm. This advantage of the PE algorithm is even greater when multiple servers are executing at different priority levels. A better algorithm for soft deadline aperiodic tasks would have the advantages of both these algorithms while overcoming their limitations. Chapter 3 describes and gives examples of such an algorithm, the *Sporadic Server* (SS) algorithm. Section 3.5 presents the results of a simulation study that compares the response time performance of the Polling, DS, PE, and SS algorithms.

One important class of aperiodic tasks that is not generally supported by previous aperiodic service algorithms consists of aperiodic tasks with hard deadlines. To guarantee hard deadlines for aperiodic tasks, a high priority server can be created with enough execution time to guarantee that an aperiodic task can meet a hard deadline. However, a minimum interarrival time restriction must be placed upon the aperiodic task to insure that the server will always have sufficient execution time to meet the aperiodic task's deadline. This minimum interarrival time for the aperiodic task must be equal to or greater than the period of its server task. As long as the aperiodic task does not request service more frequently than the minimum interarrival time, its hard deadline can be guaranteed. The SS algorithm can be used to provide such a guarantee but only if the aperiodic task's deadline is equal to or greater than its minimum interarrival time. For the cases when the deadline is shorter than the minimum interarrival time, a priority assignment that is based upon the deadline of the aperiodic task rather than its minimum interarrival time is necessary. This is referred to as a *deadline monotonic priority assignment* and requires a different schedulability analysis than that used for a rate monotonic priority assignment. The necessity of deadline monotonic priority assignment for hard deadline aperiodic tasks and the required schedulability analysis will be discussed in Chapter 4.

3. Scheduling Soft Deadline Aperiodic Tasks

In this Chapter, we investigate the scheduling of soft deadline aperiodic tasks. Our objective is to provide fast average response times for aperiodic tasks while guaranteeing the hard deadlines of periodic tasks. In this Chapter, we introduce a new algorithm, determine its schedulability bound, and compare its response time performance to previous aperiodic service algorithms.

3.1. The Sporadic Server Algorithm

Although both the DS and PE algorithms have been shown to provide good average response times for aperiodic tasks [3], these algorithms can be improved in several ways. The DS and PE algorithms have comparable aperiodic response time performance, but each algorithm has advantages over the other, as was discussed in Section 2.2. The advantage of the DS algorithm over the PE algorithm is that the DS algorithm always maintains its high priority server execution time at its original priority level. As such, the DS algorithm has a much simpler implementation than the PE algorithm because the DS algorithm does not have to manage the exchanging of aperiodic execution time between priority levels as is necessary with the PE algorithm. However, the PE algorithm has the advantage of a larger server size than the DS algorithm. A better algorithm would have an aperiodic response time performance comparable to the DS and PE algorithms, the low implementation complexity of the DS algorithm, and the larger server size of the PE algorithm. The *Sporadic Server* (SS) algorithm has these qualities.

The SS algorithm, like the DS and PE algorithms, creates a high priority task for servicing aperiodic tasks. The SS algorithm preserves its server execution time at its high priority level until an aperiodic request occurs. The SS algorithm differs from the DS and PE algorithms in the way it replenishes its server execution time. The DS and PE algorithms periodically replenish their server execution time to its full capacity. The SS algorithm only replenishes its server execution time after some or all of the execution time is consumed by aperiodic task execution. This method of replenishing server execution time sets the SS algorithm apart from the DS and PE algorithms and is central to understanding the operation of the SS algorithm.

The following terms are used to explain the SS algorithm's method of replenishing server execution time:

P_S The task priority level at which the system is currently executing.

P_i One of the priority levels in the system. Priority levels are consecutively numbered in priority order with P_1 being the highest priority level and P_2 being the next highest.

Active This term is used to describe a period of time with respect to a particular priority level. A priority level, P_i, is considered to be *active* if the current priority of the system, P_S, is equal to or higher than the priority of P_i.

Idle This term has the opposite meaning of the term *active*. A priority level, $\mathbf{P}_i$, is *idle* if the current priority of the system, $\mathbf{P}_S$, is less than the priority of $\mathbf{P}_i$.

$\mathbf{RT}_i$ The replenishment time for priority level $\mathbf{P}_i$. This is the time at which consumed execution time for the sporadic server of priority level $\mathbf{P}_i$ will be replenished.

Replenishment of consumed sporadic server execution time for a sporadic server executing at priority level $\mathbf{P}_i$ proceeds as follows:

- If the server has execution time available, the replenishment time, $\mathbf{RT}_i$, is set when priority level $\mathbf{P}_i$ becomes active. The value of $\mathbf{RT}_i$ is set equal to the current time plus the period of $\mathbf{P}_i$. If the server capacity has been exhausted, the next replenishment time can be set when the server's capacity becomes non-zero and $\mathbf{P}_i$ is active.

- Replenishment of any consumed server execution time is scheduled to occur at $\mathbf{RT}_i$, if either the priority level $\mathbf{P}_i$ becomes idle or the server's execution time has been exhausted. The amount to be replenished is equal to the amount of server execution time consumed.

3.2. SS Algorithm Example

The SS algorithm will be illustrated by comparing its operation to that of the DS and PE algorithms using a simple periodic task set. The task set is composed of two periodic tasks: τ_1 and τ_2. Task τ_1 has the shorter period and thus is assigned a higher priority than task τ_2 by the rate monotonic algorithm. For these examples both τ_1 and τ_2 begin their periods at time = 0. The periodic task set parameters are:

Task	Exec Time	Period	Utilization
τ_1	2	10	20.0%
τ_2	6	14	42.9%

For this periodic task set, the maximum server sizes were determined for the DS, PE, and SS algorithms. For each algorithm the server's period was chosen to be 5 units of time. This implies that the aperiodic server has the highest priority followed by priorities of τ_1 and τ_2. The initial periods for each of the algorithms begins at time = 0. The server size characteristics for the DS, PE, and SS algorithms are:

Algorithm	Exec Time	Period	Server Size
DS	1.00	5	20.0%
PE	1.33	5	26.7%
SS	1.33	5	26.7%

Figures 3-1, 3-2, and 3-3 show the behavior of the DS, PE, and SS algorithms for this task

set. The upper part of these figures shows the task execution order, and the lower part shows the server capacity as a function of time. In each of these examples, two aperiodic requests occur. Both requests are for aperiodic tasks that require 1.33 units of execution time. The first aperiodic request occurs at time = 1 and the second occurs at time = 8.

Figure 3-1: Deferrable Server Example

Figure 3-1 shows the behavior of the DS algorithm for this task set. At time = 0, the server's execution time is brought to its full capacity of 1.00 unit and τ_1 begins execution. The server's capacity is preserved until the first aperiodic request occurs and is serviced at time = 1. At time = 2, the server's execution time is exhausted and τ_1 resumes execution. At time = 3, τ_1 completes execution and τ_2 begins execution. At time = 5, the server's execution time is brought to its full capacity of 1.00 unit, and service for the first aperiodic request resumes, consuming 0.33 units of server execution time. At time = 5.33, the service for the first request is completed and τ_2 resumes execution. The response time for the first aperiodic request is 4.33 units of time. At time = 8, the second aperiodic request occurs and is serviced using the remaining 0.66 units of server execution time. At time = 8.66, aperiodic service is suspended and τ_2 resumes execution. At time = 10, τ_2 completes execution, the server's execution time is brought to its full capacity of 1.00 unit, and service for the second aperiodic request is resumed. At time = 10.66, service for the second aperiodic request is completed (leaving 0.33 units of server execution time) and τ_1 begins execution. The response time for the second aperiodic request is 2.66 units of time. At time = 14, τ_2 begins execution. At time = 15, the server's execution time is brought to its full capacity of 1.00 unit. At time = 20, τ_2 completes execution.

Figure 3-2 shows the behavior of the PE algorithm for this task set. Since the PE algorithm exchanges server execution time with lower priority periodic tasks, three timelines are shown in Figure 3-2 for the aperiodic time available (ATA) of priority levels 1 (the highest), 2,

Figure 3-2: Priority Exchange Example

and 3 (the lowest). At time = 0, τ_1 begins execution and the server time of priority level 1 is brought to its full capacity of 1.33 units. Since no aperiodic tasks are pending, the PE server's priority 1 execution time is exchanged with the periodic execution time of priority level 2. This exchange continues until the first aperiodic request occurs at time = 1, at which point the remaining priority 1 server time is used to service the aperiodic task. At time = 1.33, the priority 1 server time is exhausted and aperiodic service is continued using priority 2 server time. At time = 2.33, service for the first aperiodic request is complete, the PE server time is completely exhausted, and τ_1 resumes execution. At time = 3.33, τ_1 completes execution and τ_2 begins execution. At time = 5, the server time of priority level 1 is brought to its full capacity of 1.33 units. Since no aperiodic tasks are pending, the priority 1 server time is traded down to priority 3 server time by time = 6.33. At time = 8, the second aperiodic request occurs and is serviced using the priority 3 server time. At time = 9.33, service for the second aperiodic request is complete, the priority 3 server time is completely

exhausted, and τ_2 resumes execution. At time = 10, the server time of priority level 1 is brought to its full capacity of 1.33 units and τ_2 begins execution. Since no aperiodic requests occur between time = 10 and time = 20, the PE server time is exchanged for lower priority aperiodic execution time or is discarded when no exchanges are possible (as from time = 12.66 to time = 14 when no periodic or aperiodic tasks are ready to execute).

The following server characteristics should be noted by comparing the operation of the DS and PE algorithms in Figures 3-1 and 3-2.

- The DS algorithm always preserves its server execution time at its original priority level whereas the PE algorithm must exchange or discard its server execution time if no aperiodic tasks are pending. This quality of the DS algorithm allows a less complex implementation than is necessary for the PE algorithm.

- Since the PE algorithm has a larger server size, it was able to provide a better response time for the aperiodic requests.

- Both algorithms periodically replenish their server execution time at the beginning of the server period.

Figure 3-3 shows the behavior of the SS algorithm using a high priority sporadic server for this task set. Because the sporadic server is the only task executing at priority level $\mathbf{P}_1$ (the highest priority level), $\mathbf{P}_1$ becomes active only when the sporadic server executes an aperiodic task. Similarly, whenever the sporadic server is not executing an aperiodic task, $\mathbf{P}_1$ is idle. Therefore, $\mathbf{RT}_1$ is set whenever an aperiodic task is executed by the sporadic server. Replenishment of consumed sporadic server execution time will occur one server period after the sporadic server initially services an aperiodic task.

The task execution in Figure 3-3 proceeds as follows. For this example the sporadic server begins with its full execution time capacity. At time = 0, τ_1 begins execution. At time = 1, the first aperiodic request occurs and is serviced by the sporadic server. Priority level $\mathbf{P}_1$ has become active and $\mathbf{RT}_1$ is set equal to 6. At time = 2.33, the servicing of the first aperiodic request is completed, exhausting the server's execution time, and $\mathbf{P}_1$ becomes idle. A replenishment of 1.33 units of time is set for time = 6 (note the arrow in Figure 3-3 pointing from time = 1 on the task execution timeline to time = 6 on the server capacity timeline). The response time of the first aperiodic request is 1 unit of time. At time = 3.33, τ_1 completes execution and τ_2 begins execution. At time 6, the first replenishment of server execution time occurs, bringing the server's capacity up to 1.33 units of time. At time = 8, the second aperiodic request occurs and $\mathbf{P}_1$ becomes active as the aperiodic request is serviced using the sporadic server's execution time. $\mathbf{RT}_1$ is set equal to 13. At time = 9.33, the servicing of the second aperiodic request completes, $\mathbf{P}_1$ becomes idle, and τ_2 is resumed. A replenishment of 1.33 units of time is set for time = 13 (note the arrow in Figure 3-3 pointing from time = 8 on the task execution timeline to time = 13 on the server capacity timeline). At time 13, the second replenishment of server execution time occurs, bringing the server's capacity back up to 1.33 units of time.

Figure 3-3: High Priority Sporadic Server Example

By comparing Figures 3-1, 3-2, and 3-3, the following advantages of the SS algorithm are seen:

- The SS algorithm has a low implementation complexity that is comparable to the DS algorithm, because it maintains its server execution time at its original priority level and does not exchange server execution time with lower priority levels as the PE algorithm does.

- The SS algorithm has the same server size advantage over the DS algorithm as the PE algorithm does.

- The SS algorithm may also have a runtime advantage over the DS and PE algorithms. A runtime overhead is incurred periodically for the DS and PE algorithms to replenish their server execution time. This overhead is paid whether or not any aperiodic tasks were serviced during the last server period. The SS algorithm only pays a replenishment overhead if some of its server execution time has been consumed.

To better understand the SS algorithm's replenishment method, the previous high priority sporadic server example presented in Figure 3-3 is augmented with examples of an equal and a medium priority sporadic server. For these examples, presented in Figures 3-4 and 3-5, all aperiodic requests require 1 unit of execution time.

Figure 3-4 shows the task execution and the task set characteristics for the equal priority sporadic server example. The sporadic server and τ_1 both execute at priority level $\mathbf{P}_1$ and τ_2 executes at priority level $\mathbf{P}_2$. At time = 0, τ_1 begins execution, $\mathbf{P}_1$ becomes active, and $\mathbf{RT}_1$ is set to 10. At time = 2, the first aperiodic request occurs and is serviced by the sporadic server. At time = 3, service is completed for the first aperiodic request, τ_2 begins

Task	Exec Time	Period	Utilization
SS	2	10	20.0%
τ_1	2	10	20.0%
τ_2	6	14	42.9%

Figure 3-4: Equal Priority Sporadic Server Example

execution, $\mathbf{P}_1$ becomes idle, and a replenishment of 1 unit of server execution time is set for time = 10. At time = 8, the second aperiodic request occurs and is serviced using the sporadic server, $\mathbf{P}_1$ becomes active, and $\mathbf{RT}_1$ is set to 18. At time = 9, service is completed for the first aperiodic request, τ_2 resumes execution, $\mathbf{P}_1$ becomes idle, and a replenishment of 1 unit of server execution time is set for time = 18. At time = 10, τ_1 begins execution and causes $\mathbf{P}_1$ to become active and the value of $\mathbf{RT}_1$ to be set. However, when τ_1 completes at time = 12 and $\mathbf{P}_1$ becomes idle, no sporadic server execution time has been consumed. Therefore, no replenishment time is scheduled even though the priority level of the sporadic server became active.

Figure 3-4 illustrates two important properties of the sporadic server algorithm. First, $\mathbf{RT}_i$ can be determined from a time that is *less* than the request time of an aperiodic task. This occurs for the first aperiodic request in Figure 3-4 and is allowed because $\mathbf{P}_1$ became active before and remained active until the aperiodic request occurred. Second, the amount of execution time replenished to the sporadic server is equal to the amount consumed. When the PE and DS algorithms replenish their server execution time, the server capacity is always brought to its maximum value regardless of how much server execution was used.

Figure 3-5 shows the task execution and the task set characteristics for the medium

Task	Exec Time	Period	Utilization
τ_1	1.0	5	20.0%
SS	2.5	10	25.0%
τ_2	6.0	14	42.9%

Figure 3-5: Medium Priority Sporadic Server Example

sporadic server example. The sporadic server executes at priority level $\mathbf{P}_2$, between the priority levels of τ_1 ($\mathbf{P}_1$) and τ_2 ($\mathbf{P}_3$). At time = 0, τ_1 begins execution. At time = 1, τ_1 completes execution and τ_2 begins execution. At time = 4.5, the first aperiodic request occurs and is serviced using the sporadic server making priority level $\mathbf{P}_2$ active. At time = 5, τ_1 becomes active and preempts the sporadic server. At this point both priority levels $\mathbf{P}_1$ and $\mathbf{P}_2$ are active. At time = 6, τ_1 completes execution, $\mathbf{P}_1$ becomes idle, and the sporadic server is resumed. At time = 6.5, service for the first aperiodic request is completed, τ_1 resumes execution, and $\mathbf{P}_2$ becomes idle. A replenishment of 1 unit of sporadic server execution time is scheduled for time = 14.5. At time = 8, the second aperiodic request occurs and consumes 1 unit of sporadic server execution time. A replenishment of 1 unit of sporadic server execution time is set for time = 18.

Figure 3-5 illustrates another important property of the sporadic server algorithm. Even if the sporadic server is preempted and provides discontinuous service for an aperiodic request (as occurs with the first aperiodic request in Figure 3-5), only one replenishment time is necessary. Preemption of the sporadic server does not cause the priority level of the sporadic server to become idle, allowing a discontinuous consumption of sporadic server

execution time to be replenished continuously. Note that one replenishment for the consumption of sporadic server execution time resulting from both aperiodic requests in Figure 3-5 is not permitted because the priority level of the sporadic server became idle between the completion of the first aperiodic request and the initial service of the second aperiodic request.

3.3. The Schedulability of Sporadic Servers

In this section we prove that, from a scheduling point of view, a sporadic server can be treated as a standard periodic task with the same period and execution time as the sporadic server. It is necessary to prove this claim because the sporadic server violates one of the basic assumptions governing periodic task execution as described by Liu and Layland [5] in their analysis of the rate monotonic algorithm. Given a set of periodic tasks that is schedulable by the rate monotonic algorithm, this assumption requires that, once a periodic task is the highest priority task that is ready to execute, it must execute.[2] If a periodic task *defers* its execution when it otherwise could execute immediately, then it may be possible that a lower priority task will miss its deadline even if the set of tasks was schedulable. A sporadic server does not meet this Liu and Layland requirement because even though it may be the highest priority task that is ready to execute, it will preserve its execution time if it has no pending requests for aperiodic service. This preservation of the server's execution time is equivalent to deferring the execution of a periodic task. A deferrable server also fails to meet the above requirement because of its preservation capabilities. To prove that a sporadic server can still be treated as a normal periodic task, we will show that the sporadic server's replenishment method compensates for any deferred execution of the sporadic server. In contrast, we will also show how the replenishment method for a deferrable server fails in this respect.

To demonstrate how deferred execution can cause a lower priority task to miss its deadline, the execution of a periodic task will be compared to the execution of a deferrable server with the same period and execution time. Figure 3-6 presents the execution behavior of three periodic tasks. Let T represent the period of a periodic task and C represent its execution time. Task A, with $T_A = 4$ and $C_A = 1$, has the highest priority. Task B, with $T_B = 5$ and $C_B = 2$, has a medium priority. Task C, with $T_C = 10$ and $C_C = 3$, has the lowest priority. Task A and B begin their first periods at time = 0 and Task C begins its first period at time = 3. Note that no idle time exists during the first period of Task C. This constrains Task C to a maximum execution time of 3 units.

Figure 3-7 presents the execution of the task set of Figure 3-6 with Task B replaced with a deferrable server. Also presented in Figure 3-7 is the capacity of the deferrable server as a function of time. For this example, the following aperiodic requests are made to the deferrable server:

[2]For the case when two or more periodic tasks of equal priority are the highest priority tasks that are ready to execute, the requirement is that one of these tasks must execute.

Figure 3-6: Periodic Task Example

Request	Request Instant	Exec Time
1	1	1
2	3	1
3	5	2
4	10	2

Referring to the deferrable server execution graph in Figure 3-7, at time = 2 the service for the first aperiodic request completes and no other aperiodic requests are pending. At this point the deferrable server *defers* its execution by preserving its remaining execution time until the second aperiodic request occurs at time = 3. Note that this deferred execution followed by the servicing of the second aperiodic request from time = 3 to time = 4 has blocked Task C from executing during this interval; whereas during the same interval in Figure 3-6, Task C was not blocked. The third and fourth aperiodic requests are executed by the deferrable server during the same intervals as Task B executes in Figure 3-6. Thus, Task A and the deferrable server limit Task C to a maximum of 2 units of execution time per period, whereas the original periodic task was able to complete 3 units of computation during the same period in Figure 3-6. If Task C had needed 3 units of execution time during the period shown in Figure 3-7, it would have missed its deadline. It is this invasive quality of the deferrable server for lower priority periodic tasks that results in the lower scheduling bound for the DS algorithm described in Section 2.2.

Figure 3-8 presents the execution of the task set of Figure 3-6 with Task B replaced by a sporadic server. The third timeline in Figure 3-8 is the capacity of the sporadic server as a function of time. The arrows in Figure 3-8 indicate the replenishment of consumed sporadic server execution time. The requests for aperiodic service are the same as the requests of

Figure 3-7: An Example of Deferred Execution with the DS Algorithm

Figure 3-7. Note that the sporadic server, like the deferrable server in Figure 3-7, blocks the execution of Task C during time = 3 to time = 4. However, the sporadic server replenishment method prevents the execution time consumed during this interval from being replenished until time = 8. This allows Task C to execute from time = 6 to time = 7, whereas the deferrable server was executing during this interval in Figure 3-7. The sporadic server, unlike the deferrable server, blocks Task C from executing from time = 9 to time = 10. However, the replenishment of the server execution time consumed during this interval does not occur until time = 14. This allows Task C to complete its execution from time = 11 to time = 12.

Thus, in this example, the sporadic server allows Task C to meet its deadline whereas a deferrable server having the same period and execution time cannot. However, one should note that the sporadic server completes the third and fourth aperiodic requests 3 time units later than the deferrable server. Thus, for the same series of aperiodic requests, a sporadic server may provide a longer response time than an equivalently sized deferrable server. Section 3.5 presents a simulation study that investigates the relative performance of these algorithms.

Now that we have shown a specific example of how the replenishment method of the SS algorithm can compensate for deferred execution of the sporadic server, we need to prove that, in terms of schedulability, a sporadic server can always be treated as a periodic task having the same period and execution time. To do this we first show that a sporadic server

Figure 3-8: An Example of Deferred Execution with the SS Algorithm

can behave in a manner identical to a periodic task with the same period and execution time. Next we show that any execution of the sporadic server before the server's priority level becomes idle, falls into one of three cases:

1. None of the server's execution time is consumed.

2. The server's execution time is completely consumed.

3. The server's execution time is only partially consumed.

In the first case, the server can be shown to be equivalent to a periodic task with a delayed request. In the second case, the server's execution is shown to be identical to that of a normal periodic task. In the third case, the execution behavior of the server is shown to be equivalent to two periodic tasks: one that executes normally and one that is delayed. Thus, all types of sporadic server execution are shown to be equivalent to the execution of one or more periodic tasks.

To explore the schedulability effects of sporadic servers upon a periodic task set, several terms and concepts developed by Liu and Layland in [5] are needed. A periodic task set is composed of *n* independent tasks, τ_1 through τ_n, numbered in order of decreasing priority. A periodic task, τ_i, is characterized by its computation time, C_i and a period, T_i. At all times the highest priority task that is ready to execute is selected for execution, preempting any lower priority tasks as necessary. The deadline of a periodic task occurs one period after it

is requested. The *critical instant* for a periodic task is the instant at which a request for that task will have the longest response time. Liu and Layland [5] prove the following theorem concerning the critical instant of a periodic task:

> **Theorem 1:** Given a set of periodic tasks, the critical instant for any task occurs whenever the task is requested simultaneously with all higher priority tasks.

The *critical zone* for a periodic task is the time interval between its critical instant and the completion of that request. A periodic task set is schedulable if the critical zone for each task is less than or equal to its deadline.

We now prove the following property of schedulable periodic tasks sets:

> **Lemma 2:** Given a periodic task set that is schedulable with τ_i, the task set is also schedulable if τ_i is split into *k* separate periodic tasks, $\tau_{i,1}, \ldots, \tau_{i,k}$, with $\sum_{j=1}^{k} C_{i,j} = C_i$ and $T_{i,j} = T_i$ for $1 \leq j \leq k$.
>
> **Proof:** By the rate monotonic algorithm, all tasks $\tau_{i,1}, \ldots, \tau_{i,k}$ will be assigned the same priority because they have the same period. The rate monotonic algorithm schedules a periodic task set independently of the phasing between any two tasks. Suppose the requests for tasks $\tau_{i,1}, \ldots, \tau_{i,k}$ are all in sync with each other. The execution pattern of $\tau_{i,1}, \ldots, \tau_{i,k}$ will now be identical to the execution pattern of the original task, τ_i, and therefore the task set is still schedulable for this phasing of $\tau_{i,1}, \ldots, \tau_{i,k}$. Since the task set is schedulable for one phasing of $\tau_{i,1}, \ldots, \tau_{i,k}$ it is schedulable for them all. The Lemma follows.

Next we establish that the execution behavior of a sporadic server can be identical to that of a periodic task with the same period and execution time.

> **Lemma 3:** Given a schedulable periodic task set, replace a periodic task with a sporadic server having the same period and execution time. If the requests for the sporadic server are identical to that of the original periodic task, then the execution behavior of the sporadic server is identical to that of the original periodic task.
>
> **Proof:** The periodic task and the sporadic server execute at the same priority level, and each request for the sporadic server and the periodic task require the same execution time. Each request for the sporadic server completely consumes its execution time, and the consumed execution time is replenished before the next request. Therefore, the sporadic server's execution is identical to that of the original periodic task.

We now show that sporadic servers are equivalent to periodic tasks in terms of schedulability.

> **Theorem 4:** A periodic task set that is schedulable with a task, τ_i, is also schedulable if τ_i is replaced by a sporadic server with the same period and execution time.
>
> **Proof:** To prove this theorem we will show that, for all types of sporadic server execution, the sporadic server exhibits an execution behavior that can also be represented by a combination of one or more periodic tasks.

Let the sporadic server's period be T_{SS} and execution time be C_{SS}. At time t_A, let the priority level of the server become active and let the sporadic server be at full capacity. Consider the interval of time beginning at t_A during which the priority level of the server remains active. The execution behavior of the server during this interval can be described by one of three cases:

1. None of the server's execution time is consumed.

2. All of the server's execution time is consumed.

3. The server's execution time is partially consumed.

Case 1: If the server is not requested during this interval, it preserves its execution time. The server's behavior is identical to a periodic task for which the interarrival time between two successive requests for the periodic task is greater than its period. Referring to Theorem 1, if any request for a periodic task is delayed, it cannot lengthen critical zone of any lower priority periodic task because the lower priority tasks would be able to execute during the delay, yielding a *shorter* critical zone. Since a critical zone of a lower priority task cannot be made longer, the delay has no detrimental schedulability effect.

Case 2: If the execution time of the sporadic server is completely consumed, a replenishment will be scheduled to occur at $t_A + T_{SS}$ to bring the server back to full capacity. By Lemma 3 the behavior of the sporadic server between t_A and $t_A + T_{SS}$ is identical to that of a similar periodic task that is requested at t_A.

Case 3: If the server's execution time is not completely exhausted, a replenishment will be scheduled to occur at $t_A + T_{SS}$ for the consumed execution time and the server will preserve the unconsumed execution time until it is requested. Let the amount of execution time to be replenished be C_R.

Now consider a periodic task with a period of T_{SS} and an execution time of C_{SS} that is split into two periodic tasks, τ_x and τ_y, both with a period of T_{SS}. Let τ_x have an execution time of C_R and let τ_y have an execution time of $C_{SS} - C_R$. By Lemma 2, the splitting of the original periodic task can be done without affecting schedulability. Let requests for both τ_x and τ_y be in sync until t_A. At t_A, let τ_x execute normally, but delay τ_y. As in Case 1, the delay for τ_y has no schedulability effect. The behavior of the two periodic tasks τ_x and τ_y from t_A to $t_A + T_{SS}$ is identical to that of the sporadic server over the same time interval.

Because a sporadic server's execution in each of these cases can be represented by a periodic task or a combination of periodic tasks with a period and total execution time that is identical to that of the sporadic server, the Theorem follows.

If a sporadic server's execution time is only partially consumed before a replenishment is scheduled, as described in Case 3 of the proof for Theorem 4, the server must then manage two quantities of execution time: the execution time that will be replenished and the unconsumed execution time. This could be a concern for sporadic server implementations if many successive executions of the sporadic server split the server's execution time into many small quantities. However, if the server is not requested for a while, these small quantities can be merged together. In fact, regardless of how many times a server's execution time has been split into smaller quantities, if the server is not requested for an amount of time equal to or greater than its period, then all of these quantities can be merged together. An example of this behavior is shown in Figure 3-9. The sporadic server is the highest priority task in the system; it has a period of 5 and an execution time of 2. The first four requests for the server require 0.5 units of execution time. Because the priority level of the sporadic server becomes idle between each of these requests, the execution time of the sporadic server has been split four ways, requiring four separate replenishment times. However, no requests are made to the server from time = 4 to time = 9, one server period after the last request. By time = 9, the server's capacity has been completely replenished and merged together.

Figure 3-9: An Example of The Splitting and Merging of Sporadic Server Execution Time

In this Chapter we have shown that, although a sporadic server can defer its execution time, it can still be treated as a periodic task. However, it was also shown that this is not true for a deferrable server. The key difference between these two types of servers is that when a sporadic server defers its execution, it also defers the replenishment of its execution time once it is consumed. In contrast, a deferrable server replenishes its execution time independently of when its execution time is consumed.

3.4. Sporadic Servers and Priority Inheritance Protocols

In this Chapter we define the interaction of sporadic servers and the priority inheritance protocols developed by Sha, Rajkumar, and Lehoczky in [8].

Mok [6] has shown that the problem of determining the schedulability of a set of periodic tasks that use semaphores to enforce exclusive access to shared resources is NP-hard. The semaphores are used to guard critical sections of code (e.g., code to insert an element into a shared linked list). To address this problem for rate monotonic scheduling, Sha, Rajkumar, and Lehoczky [8] have developed priority inheritance protocols and derived sufficient conditions under which a set of periodic tasks that share resources using these protocols can be scheduled. The priority inheritance protocols require that the priority of a periodic task be temporarily increased if it is holding a shared resource that is needed by a higher priority task. This technique of temporarily increasing a task's priority is also the basis for the implementation of sporadic servers. Since both sporadic servers and the priority inheritance protocols manipulate the priorities of tasks, it is necessary to define the interaction of these two scheduling techniques.

The priority inheritance protocols manipulate the priorities of tasks that enforce mutual exclusion using semaphores in the following manner.[3] Consider the case of a task, τ_A, that is currently executing and wants to lock a semaphore and enter a critical section. The priority inheritance protocols will select one of the following two sequences of task execution:

1. Task τ_A is allowed to lock the semaphore and enter the critical section. During the critical section τ_A executes at its assigned priority.

2. Task τ_A is not allowed to lock the semaphore and is blocked from executing. The lower priority task that is causing the blocking then inherits the priority of τ_A and continues execution.

We are concerned with the problem of the interaction of priority inheritance protocols and a sporadic server for the case when an aperiodic task wants to lock a semaphore and enter a critical section. The following rules govern the interaction of the priority inheritance protocols and the use of sporadic servers:

1. The execution of an aperiodic task that is using its sporadic server consumes the sporadic server's execution time. Similarly, the execution of any task that inherits the priority of a sporadic server from an aperiodic task also consumes the sporadic server's execution time.

2. When a sporadic server's execution time is exhausted, all tasks that have inherited the priority of the sporadic server should then return to the priority they had before inheriting the sporadic server's priority.

[3] The description here of the operation of the priority inheritance protocols is very simplistic but sufficient for describing the interaction of sporadic servers and the priority inheritance protocols. For a better description of the priority inheritance protocols, the reader is referred to [8].

Consider the simple example of an aperiodic task, τ_A, that is using its sporadic server but whose execution is being blocked by a low priority task, τ_L, that has locked a semaphore. By the priority inheritance protocols and rule 1, τ_L will inherit the priority of the sporadic server. The execution of τ_L until it releases the semaphore will consume the sporadic server's execution time. If the sporadic server's execution time were exhausted before τ_L released the semaphore, the priority of τ_A would return to its assigned priority and, by rule 2, τ_B would return to the priority it had before inheriting the priority of the sporadic server.

The determination of schedulability for soft deadline aperiodic tasks that use a sporadic server is dependent upon whether or not the aperiodic task uses semaphores to share data with other tasks. If the aperiodic task does not share data with other tasks, then schedulability analysis for the sporadic server is sufficient. However, if the aperiodic task does share data, a different analysis is required. Consider the case where an aperiodic task is using its sporadic server's execution capacity and locks a semaphore. During the execution of the critical section, the sporadic server's capacity may be exhausted, forcing the aperiodic task to return to background priority. If this happens, the aperiodic task can potentially block any higher priority task that wants to lock the semaphore. Thus, the critical sections of this aperiodic task can become the blocking time for any higher priority tasks that may want to lock the semaphore. This blocking time can be accounted for by using the schedulability analysis equations developed in [8]. We will also discuss the use of these equations for scheduling short deadline aperiodic tasks in Chapter 4.

3.5. Sporadic Server Performance

Simulations were conducted to compare the response time performance of the Polling, DS, PE, and SS algorithms for soft deadline aperiodic tasks. For these simulations, a set of ten periodic tasks with periods ranging from 54 to 1200 was chosen. Three periodic loads were simulated for this set of periods: 40%, 69%, and 88%. The 40% load represents a low periodic load. The 69% case corresponds to the maximum resource utilization for which the rate monotonic algorithm can always guarantee schedulability, independent of the particular task periods [5]. The 88% case represents the average case scheduling bound for the rate monotonic algorithm [2]. The aperiodic load for these simulations was varied across the range of resource utilization unused by the periodic tasks. The interarrival times for the aperiodic tasks were modeled using a Poisson arrival pattern. There are three different average interarrival times: 18, 36, and 54. The aperiodic service times were modeled using an exponential distribution.

The server periods for each of the Polling, DS, PE, and SS algorithms were chosen so that the aperiodic server would have the highest priority in the system. The server periods for the Polling, PE, and SS algorithms were set equal to the shortest periodic task. Since priority ties are broken in favor of aperiodic tasks, this period assignment gives these aperiodic servers the highest priority. The server execution times for these algorithms were selected to be the maximum value for which all the periodic tasks remain schedulable. The periods and execution times for the DS algorithm were chosen in a different manner. In

[11], guidelines were given for selecting the period for the deferrable server in order to avoid a worst case utilization bound for periodic tasks. The guideline suggests that $T_{DS} \leq T_{min} / (1 + U_{DS})$ where T_{DS} is the deferrable server period, T_{min} is the smallest period of the periodic tasks, and U_{DS} is the utilization of the deferrable server. For each periodic load, the maximum deferrable server utilization for which all the periodic tasks remain schedulable for $T_{DS} = 27$ was found. This utilization value was then used in the above formula to obtain a new value for T_{DS}. The resulting deferrable server periods were 36 for a periodic load of 40%, 44 for a periodic load of 69%, and 52 for a periodic load of 88%. For each periodic load, the maximum deferrable server utilization for which all the periodic tasks remain schedulable was found using these new values for T_{DS}.

Figures 3-10, 3-11, and 3-12 present the results of these simulations for each of the mean aperiodic interarrival times simulated (18, 36, and 54). In each of these figures, three graphs are presented, which correspond to the different periodic loads simulated (40%, 69%, and 88%). In all of the graphs of these figures, the average aperiodic response time of each algorithm is plotted as a function of the aperiodic load. The average aperiodic response time of each algorithm is presented relative to the response time of background aperiodic service. In other words, the average response time equivalent to background service has a value of 1.0 on all the graphs. An improvement in average aperiodic response time over background service corresponds to a value of less than 1.0. The lower the response time curve lies on these graphs, the better the algorithm is for improving aperiodic responsiveness.

As can be seen from each of the graphs presented in Figures 3-10, 3-11, and 3-12, the DS, PE, and SS algorithms can provide a substantial reduction in average aperiodic response time compared to background or polling aperiodic service. These algorithms provide the greatest improvement for short, frequent aperiodic tasks as can be seen in the lower left-hand side of each graph. The performance of the SS algorithm in each of these graphs is comparable to the performance of the DS and PE algorithms.

Figure 3-10: SS Algorithm Performance Comparison,
Mean Aperiodic Interarrival Time = 18

Figure 3-11: SS Algorithm Performance Comparison,
Mean Aperiodic Interarrival Time = 36

Figure 3-12: SS Algorithm Performance Comparison,
Mean Aperiodic Interarrival Time = 54

Referring to the 40% periodic load case presented in graph *(a)* of Figures 3-10, 3-11, and 3-12 the performance of the SS and PE algorithms becomes better than the performance of the DS algorithm as the aperiodic load increases. The performance of the DS algorithm even becomes worse than the performance of Polling for high aperiodic load. This increase in the relative response time for the DS algorithm is due to its smaller server size of 45.3% compared to a server size of 56.3% for the Polling, PE, and SS algorithms. The larger server size of the SS algorithm is an advantage over the DS algorithm for a task set with a moderate periodic load and a high aperiodic load.

The 69% and 88% periodic load cases presented in graphs *(b)* and *(c)* of Figures 3-10, 3-11, and 3-12 show that the average response time of the SS algorithm becomes slightly higher than the response time of the DS and PE algorithms for moderate to high aperiodic load. The behavior is not attributable to a difference in server size, because as the periodic load increases, the difference in server sizes diminishes. The difference between the DS server size and server size for the Polling, PE, and SS algorithms is about 1% for a periodic load of 69% and decreases to about 0.1% for a periodic load of 88%. The cause for the slightly higher response time of the SS algorithm is due to its more restrictive replenishment method as discussed in Section 3.3. The DS and PE algorithms periodically replenish their server execution time independent of when the server's execution time was consumed. The SS algorithm replenishes consumed server execution time in a sporadic manner that depends upon when the execution time is consumed. On the average, once server execution time is exhausted, aperiodic tasks must wait longer for the SS algorithm to replenish its server execution time than for the DS or PE algorithms.

4. Scheduling Sporadic Tasks

One important type of aperiodic task that is not generally supported by previous aperiodic service algorithms is a sporadic task, which is an aperiodic task with a hard deadline and a minimum interarrival time. To guarantee a hard deadline for a sporadic task, a high priority server can be created to exclusively service the sporadic task. This server preserves its execution time at its original priority level until it is needed by the sporadic task. To guarantee that the server will always have sufficient execution time to meet the sporadic task's deadline, an minimum interarrival time restriction must be placed upon the sporadic task. This minimum interarrival time must be equal to or greater than the period of the server task. As long as the sporadic task does not request service more frequently than the minimum interarrival time, its hard deadline can be guaranteed. The SS algorithm can be used to provide a guarantee for a sporadic task as long as the aperiodic task's deadline is equal to or greater than its minimum interarrival time. However, for the cases when the deadline is shorter than the minimum interarrival time, a different priority assignment for the sporadic server and a different schedulability analysis are necessary. The priority of the sporadic server should be based upon the deadline of its associated sporadic task, not upon the maximum arrival rate of the sporadic task. This type of priority assignment, first considered by Leung and Whitehead in [4], is referred to as deadline monotonic and requires a schedulability analysis different from that necessary for a rate monotonic priority assignment. Apart from the assignment of server priority, the operation of a deadline monotonic sporadic server is identical to that of the SS algorithm as presented in Section 3.1. This section demonstrates the necessity of a deadline monotonic priority assignment for a short deadline sporadic task and describes the schedulability analysis for a task set where the priorities of the periodic tasks are assigned by the rate monotonic algorithm and the priorities of the sporadic servers are assigned with the deadline monotonic algorithm.

4.1. A Simple Example of the Deadline Monotonic Priority Assignment

The necessity for a deadline monotonic priority assignment for a short deadline sporadic task will be illustrated with a simple example that shows how a rate monotonic priority assignment cannot guarantee the sporadic task's hard deadline while a deadline monotonic priority assignment can be used to guarantee that the hard deadline will always be met. The set of tasks presented in Figure 4-1 will be used for this example.

Figure 4-2 presents the execution of these tasks using the rate monotonic priority assignment for the sporadic task. In Figure 4-2 a high priority is represented by a low number. Note that the rate monotonic algorithm assigns the lowest priority to the sporadic task because of its long minimum interarrival time. The task execution graph presented in Figure 4-2 assumes that requests for both periodic tasks and the sporadic task occur at time = 0. Both periodic tasks meet their deadlines, but the sporadic task completes only 2 of the required 8 units of execution time before it misses its deadline. Clearly, the rate monotonic algorithm is inappropriate for this task set.

Figure 4-1: Periodic Task Set with Short Deadline Aperiodic Task

Figure 4-2: Example of Rate Monotonic Priority Assignment for a Short Deadline Sporadic Task

Because a higher priority is needed for the short deadline sporadic task, one might consider treating it as a periodic task with a period of 10. This would give the sporadic task the highest priority and guarantee that its deadline would be met. However, this would cause

other problems. The sporadic task only needs a resource utilization of 25% to meets its deadline. Treating it as a periodic task with a period of 10 means that a resource utilization of 80% (an execution time of 8 units divided by a period of 10 units) would be dedicated to a task that needs only 25%. This is a very wasteful scheduling policy. Also, if the priority of the sporadic task is increased in this manner, the total utilization required to schedule all three tasks is now 133.3% (33% + 20% + 80%) making this an infeasible schedule. A different method is needed that will assign priorities such that all their deadlines will be met using an efficient level of scheduled resource utilization.

A deadline monotonic priority assignment can be used to efficiently schedule the task set presented in Figure 4-1. A sporadic server is created with an execution time of 8 units and a period of 32 units to service the short deadline sporadic task. The priority of the sporadic task is assigned based upon the deadline of the task it is servicing, not upon its period. In other words, the rate monotonic algorithm is used to assign priorities for the periodic tasks, and the priority of the sporadic task is assigned as if its rate of occurrence were equal to its deadline. The task execution graph for this priority assignment is presented in Figure 4-3. The sporadic task now has the highest priority and meets its deadline. The execution of the periodic tasks is delayed, but both still meet their deadlines.

Figure 4-3: Example of Deadline Monotonic Priority Assignment for a Short Deadline Sporadic Task

Now that we have shown that a deadline monotonic priority assignment can be used to guarantee that a sporadic task will meet its deadline, we need to be able to perform a

schedulability analysis that will indicate whether or not a deadline monotonic priority assignment for the sporadic server and a rate monotonic priority assignment for the periodic tasks is feasible for a given task set. The deadline monotonic priority assignment raises the priority of the sporadic server above the priority that would be assigned by the rate monotonic algorithm. As such, the sporadic server can be considered as a low priority task that is allowed to block a higher priority task from executing. A similar problem can occur when periodic tasks that share data using critical sections are scheduled using the rate monotonic algorithm (as was seen earlier in Section SS-PCP). A low priority task that has already entered a critical section can block the execution of a high priority task that needs to enter the critical section. The blocking of the high priority task continues as long as the critical section is locked by the lower priority task. Sha, Rajkumar, and Lehoczky [8] have developed the priority ceiling protocol for a set of periodic tasks that share data using critical sections. This protocol establishes a set of sufficient conditions under which the periodic tasks can be scheduled using the rate monotonic algorithm. The schedulability analysis equations developed for the priority ceiling protocols can be used to test the schedulability of sporadic servers with deadline monotonic priority assignments. These equations are presented below:

(1)

$$\forall\ i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1).$$

$$\forall\ i,\ 1 \le i \le n, \qquad \min_{(k,\,l) \in R_i} \left[\sum_{j=1}^{i-1} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \right] \le 1 \tag{2}$$

where C_i and T_i are respectively the execution time and period of task τ_i, $U_i = C_i/T_i$ is the utilization of task τ_i, and $R_i = \{ (k, l) \mid 1 \le k \le i, l = 1, \cdots, \lfloor T_i/T_k \rfloor \}$. The term B_i is the worst-case blocking time for task τ_i.

Equations 1 and 2 were derived from equations developed for testing the schedulability of a set of periodic tasks whose priorities have been assigned using the rate monotonic algorithm. Equation 1 was derived using the worst case utilization bound equation for scheduling periodic tasks developed by Liu and Layland in [5], which, under the absolute worst case conditions, provides a sufficient condition for determining schedulability of a rate monotonic priority assignment. Equation 2 was derived from an equation developed by Lehoczky, Sha, and Ding [2] that provides necessary and sufficient conditions for determining schedulability of a rate monotonic priority assignment. Both Equations 1 and 2 provide sufficient conditions for scheduling a task set where high priority tasks can be blocked by lower priority tasks. However, Equation 1 represents the absolute worst case conditions, and a much tighter characterization is provided by Equation 2.

The blocking term, B_i, in Equations 1 and 2 represents the amount of time that a lower priority task can block a higher priority task from executing. For a sporadic server with a deadline monotonic priority, B_i is used to represent the amount of time the sporadic server

can block a periodic task that has a higher priority than the rate monotonic priority of the sporadic server. To check the schedulability of a periodic task set with a deadline monotonic sporadic server, the term B_i is set equal to the execution time of the sporadic server for all τ_i with a priority less than or equal to the priority of the sporadic server and greater than the sporadic server's original rate monotonic priority. For all τ_i with a priority less than the sporadic server's original rate monotonic priority, the sporadic server should be treated normally (i.e., treated as a normal sporadic server with a rate monotonic priority assignment). For all τ_i with a priority greater than the priority of the sporadic server, the corresponding value of B_i is set to zero.

The use of Equations 1 and 2 will be demonstrated with the task set presented in Figure 4-1. Let τ_1 and τ_2 be the periodic tasks and let τ_3 be the short deadline sporadic task. Below are the parameters for each task:

Task	C_i	T_i	B_i
τ_1	4	12	8
τ_2	4	20	8
τ_3	8	32	0

Evaluation of Equation 1 proceeds as follows:

$$i = 1, \quad \frac{C_1}{T_1} + \frac{B_1}{T_1} \leq 1$$

$$\frac{4}{12} + \frac{8}{12} \leq 1$$

$$\frac{12}{12} \leq 1$$

$$i = 2, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} \leq 2(2^{1/2}-1).$$

$$\frac{4}{12} + \frac{4}{20} + \frac{8}{20} \leq 0.824$$

$$\frac{56}{60} \geq 0.824$$

The inequality for i = 1 holds, but the inequality for i = 2 does not; therefore, it is not necessary to check the i = 3 case, and Equation 2 must now be used. The evaluation of Equation 2 proceeds as follows.

i = 1: Check if $C_1 + B_1 \leq T_1$. Since $4 + 8 \leq 12$, task τ_1 is schedulable.

i = 2: Check whether either of the following two inequalities hold:

(k, l)		
(1, 1)	$C_1 + C_2 + B_2 \leq T_1$	$4 + 4 + 8 \geq 12$
(2, 1)	$2C_1 + C_2 + B_2 \leq T_1$	$8 + 4 + 8 \leq 20$

The inequality for (k, l) = (2, 1) holds and therefore, τ_2 is schedulable.

i = 3: Check whether any of the following inequalities hold:

(k, l)		
(1, 1)	$C_1 + 2C_2 + C_3 + B_3 \leq T_1$	$4 + 8 + 8 + 0 \geq 12$
(1, 2)	$2C_1 + 2C_2 + C_3 + B_3 \leq 2T_1$	$8 + 8 + 8 + 0 \leq 24$
(2, 1)	$2C_1 + C_2 + C_3 + B_3 \leq 8T_2$	$8 + 4 + 8 + 0 \leq 20$
(3, 1)	$3C_1 + 2C_2 + C_3 + B_3 \leq 8T_2$	$12 + 8 + 8 + 0 \leq 32$

Since all the inequalities except (k, l) = (1, 1) hold, τ_3 is schedulable. Note that it would have been sufficient to stop checking after finding one inequality that holds for each value of i; all the inequalities are listed here for completeness only.

5. Summary and Conclusions

This paper described the Sporadic Server (SS) algorithm, a general algorithm for scheduling soft and hard deadline aperiodic tasks in real-time systems. The SS algorithm creates a server with a given period and utilization. The server maintains its utilization until it is needed by an aperiodic task and replenishes any consumed utilization in a sporadic manner based upon when the utilization is consumed. For soft deadline aperiodic tasks, a high priority sporadic server is created that is shared by the soft deadline aperiodic tasks. For hard deadline aperiodic tasks (sporadic tasks), an individual sporadic server is dedicated to each hard deadline aperiodic task to guarantee that its deadline will always be met.

The SS algorithm was shown to have the low implementation complexity advantage of the Deferrable Server (DS) algorithm and the server size advantage of the Priority Exchange (PE) algorithm. The SS algorithm also provides a response time performance for soft deadline aperiodic tasks that is comparable to that attainable with either the DS or PE algorithms. In terms of schedulability, it was shown that a sporadic server can be treated as a periodic task having the same period and execution time. For short deadline sporadic tasks, it was also shown that the SS algorithm can guarantee their deadlines by using a deadline monotonic priority assignment rather than a rate monotonic priority assignment. Finally, it was shown that the schedulability analysis formulas developed for the priority ceiling protocols can be used to determine the schedulability of deadline monotonic sporadic servers.

Acknowledgments

The authors would like to thank Ragunathan Rajkumar for his comments and help with this work.

References

Table of Contents

List of Figures