The Journal of Real-Time Systems, 1, 243-264 (1989)

Mode Change Protocols for Priority-Driven Preemptive Scheduling

LUI SHA
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213

RAGUNATHAN RAJKUMAR
Department of Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213

JOHN LEHOCZKY
Department of Statistics, Carnegie Mellon University, Pittsburgh, PA 15213

KRITHI RAMAMRITHAM
Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003

Abstract. In many real-time applications, the set of tasks in the system, as well as the characteristics of the tasks, change during system execution. Specifically, the system moves from one mode of execution to another as its mission progresses. A mode change is characterized by the deletion of some tasks, addition of new tasks, or changes in the parameters of certain tasks, for example, increasing the sampling rate to obtain a more accurate result. This paper discusses how mode changes can be accommodated within a given framework of priority driven real-time scheduling.

1. Introduction

To successfully develop a large-scale real-time system, we must be able to manage both the logical complexity and timing complexity by using a disciplined approach. The logical complexity is addressed by software engineering methodology, while the timing complexity is addressed by research in real-time scheduling algorithms (Lehoczky and Sha 1986; Lehoczky, Sha and Strosnider 1987; Leinbaugh 1980; Leung and Merrill 1980; Liu and Layland 1973; Rajkumar, Sha and Lehoczky 1987; Ramamritham and Stankovic 1984; Sha, Rajkumar and Lehoczky 1987; Zhao, Ramamritham and Stankovic 1987). An important class of scheduling algorithms is known as static priority scheduling algorithms. These algorithms have several attractive properties. First, they are simple to implement. Second, they have good performance. In a uni-processor, the CPU utilization bound of a randomly chosen periodic task set is 88 percent (Lehoczky, Sha and Ding 1987), while the worst-case bound for any task set is 69 percent (Liu and Layland 1973). In many applications, periodic tasks are often harmonic or nearly harmonic, and this leads to utilization bounds at or near 100 percent.

In addition to good performance found in practice, static priority scheduling algorithms are analyzable for a wide variety of practical conditions such as: (a) the scheduling of a mixture of periodic and aperiodic tasks (Lehoczky, Sha and Strosnider 1987), (b) the handling of transient overloads (Sha, Lehoczky and Rajkumar 1986), and (c) the effect of using semaphores (Sha, Rajkumar and Lehoczky 1987) and Ada rendezvous (Goodenough and Sha 1986) for task synchronization. From a software engineering point of view, these algorithms translate complex timing constraints into simple resource utilization constraints. As long as the utilization constraints of the CPU, I/O channels and communication media are observed, the deadlines of periodic tasks and the response time requirements of aperiodic tasks will both be met (Lehoczky and Sha 1986). This means that the real-time software can be modified freely as long as the utilization bounds are observed. Furthermore, should there be a transient overload, the tasks that will miss deadlines will miss them in reverse order of importance, and the number of tasks missing their deadlines will be a function of the overload (Sha, Lehoczky and Rajkumar 1986).

However, in many applications neither the task set nor the task parameters can remain static throughout the mision. A change in operational mode often leads to the modification of task parameters (for example, task period and execution time) as well as the addition of some new tasks and deletion of some existing tasks. For example, a phase array radar can adjust its sampling rate for the tracking task. Generally speaking, there are two types of mode change issues: application issues and runtime management issues. Application issues deal with the semantics of mode change: the condition for initiating a mode change, the set of tasks to be replaced or modified, and the sequence to delete, add and modify tasks. In this paper, we do not address the application issues of mode change. We assume that when a mode change is initiated, we are given a list of tasks to be modified, added or deleted, and the sequence to do so.

The focus of this paper is on the runtime management of the mode change process. Specifically, we focus upon the scheduling of mode change activities and of tasks during the transition period of mode change. Our objective is to accomplish the mode change process quickly, subject to keeping the consistency of shared data and to meeting the deadlines of tasks that must execute before, during, and after a mode change. This paper is intended to provide an overview on rate-monotonic based scheduling methods and to show how mode changes can be accommodated within this framework. In addition, we analyze the time delay associated with using the mode change protocol presented in this paper.

This paper is organized as follows. In Section 2, we first review the rate-monotonic algorithm and the priority ceiling protocol for scheduling periodic tasks since our mode change protocol will be designed to be compatible with them. In Section 3, we develop the basic mode change protocol and analyze the properties of the basic protocol. In Section 4, we first examine some possible alternatives to the basic protocol. Next, we consider the interplay between this basic protocol and other scheduling issues, namely, the period transformation method for maintaining stability under transient overload and the server algorithms for scheduling both periodics and aperiodics. Finally in Section 5, we present the concluding remarks.

2. Scheduling periodic tasks

In this section, we first review the rate-monotonic scheduling algorithm for independent periodic tasks and then review the priority ceiling protocol designed for the synchronization of periodic tasks using the rate-monotonic scheduling approach. We shall first define the basic concepts and state our assumptions before presenting a review of the scheduling algorithms. A *job J* with execution time C is a sequence of instructions that will continuously use the processor until its completion if it is executing alone on the processor. That is, we assume that a job J does not suspend itself, say for I/O operations; however, such a situation can be accommodated by defining two or more jobs. In addition, we assume that the critical section of a job is *properly* nested, that is, semaphores will be unlocked in the reversed order of locking. A job will release all of its locks, if it holds any, before or at the end of its execution. In all our discussions below, we assume that jobs $J_1, J_2, \ldots, J_n$ are listed in descending order of priority with J_1 having the highest priority. A *periodic task* τ is a sequence of the same type of job J occurring at regular intervals, $\{kT, k = 0, 1, 2, \ldots\}$, where T is the period of task τ. An *aperiodic task* is a sequence of the same type of job occurring at irregular intervals. Each task is assigned a fixed priority P, and every job of the same task is initially assigned that task's priority. If several jobs are eligible to run, the highest priority job will be run. Jobs with the same priority are executed according to a first-come first-serve discipline. When a job J is forced to wait for the execution of lower priority jobs, job J is said to be *blocked*. When a job waits for the execution of higher priority jobs or equal priority jobs that have arrived earlier, it is not considered blocked.

In the following, we first review the scheduling of independent periodic tasks. Next, we review the synchronization of periodic tasks and illustrate the issues with an example.

2.1. Scheduling independent periodic tasks

From a scheduling point of view, tasks are considered independent if they do not need to synchronize their executions with each other. Given a set of independent tasks, the scheduler can always preempt the execution of a lower priority task whenever a high priority task is ready to execute. Given a set of independent periodic tasks, the rate-monotonic scheduling algorithm gives a fixed priority to each task and assigns higher priorities to tasks with shorter periods. A task set is said to be *schedulable* if all its deadlines are met, that is, if every periodic task finishes its execution before the end of its period. Any set of independent periodic tasks is schedulable by the rate-monotonic algorithm if the condition of Theorem 1 is met (Liu and Layland 1973).

THEOREM 1: A set of n independent periodic tasks scheduled by the rate-monotonic algorithm will always meet its deadlines, for all task phasings, if

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

where C_i and T_i are the execution time and period of task τ_i respectively.

Theorem 1 offers a sufficient (worst-case) condition that characterizes the schedulability of the rate-monotonic algorithm. This bound converges to 69 percent (*ln* 2) as the number of tasks approaches infinity. Table 1 shows values of the bound for 1 to 10 tasks.

Table 1. Worst-case scheduling bounds as a function of number of tasks.

Scheduling Bounds	
Number of Tasks	Utilization Bound
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
7	0.728
8	0.724
9	0.720
10	0.718

The utilization bound of Theorem 1 is pessimistic because the worst-case task set is contribed and rather unlikely to be encountered in practice. For a randomly chosen task set, the likely bound is 88 percent (Lehoczky, Sha and Ding 1987). To know if a set of given tasks with utilization greater than the bound of Theorem 1 can meet its deadlines, the conditions of Theorem 2 can be checked (Lehoczky, Sha and Ding 1987).

THEOREM 2: A set of n independent periodic tasks scheduled by the rate-monotonic algorithm will always meet its deadlines, for all task phasings, if and only if

$$\forall i,\ 1 \le i \le n,\ \min_{(k,l)\in R_i} \sum_{j=1}^{i} C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \le 1$$

where C_j and T_j are the execution time and period of task τ_j respectively and $R_i = \{(k, l) \mid 1 \le k \le i,\ l = 1, \ldots \lfloor T_i/T_k \rfloor \}$.

Theorem 2 provides the exact criterion for testing the schedulability of independent periodic tasks using the rate-monotonic algorithm. In effect, the theorem checks if each task can complete its execution before its first deadline by checking all the scheduling points.[1] The scheduling points for task τ are τ's first deadline and the end of periods of higher priority tasks within τ's first deadline. In each application of the formula, i corresponds to the task τ_i whose deadline is to be checked, and k corresponds to each of the tasks that affects the completion time of task τ, that is, task τ_i itself and the higher priority tasks. For given i and k, l represents the scheduling points of task τ_k. For example, suppose that we have tasks τ_1 and τ_2 with periods $T_1 = 5$ and $T_2 = 14$. For task (τ_i, $i = 1$) we have only *one* scheduling point, the end of task τ_1's first period, that is, $i = k = 1$ and ($l = 1, \ldots, \lfloor T_i/T_k \rfloor = \lfloor T_1/T_1 \rfloor = 1$). The scheduling point is, of course, τ_1's first deadline ($lT_k = 5$, $l = 1$, $k = 1$). For task (τ_i, $i = 2$), there are *two* scheduling points from all higher priority tasks, (τ_k, $k = 1$), that is, ($l = 1, \ldots, \lfloor T_i/T_k \rfloor = \lfloor T_2/T_1 \rfloor = 2$). The two scheduling points correspond to the two end-points of task τ_1's period within the first deadline of task τ_2 at 14, that is, ($lT_k = 5, l = 1, k = 1$) and ($lT_k = 10, l = 2, k = 1$). Finally, there is the scheduling point from τ_2's own first deadline, that is, ($lT_k = 14, l = 1, k = 2$). At each scheduling point, we check if the task in question can complete its execution at or before the scheduling point. A detailed illustration of the application of this theorem and its generalization is given in Example 3 in Section 2.3.

2.2. Task synchronization

In the previous sections we have discussed the scheduling of independent tasks. Tasks, however, do interact and hence need to be synchronized. Common synchronization primitives include semaphores, locks, monitors, and Ada rendezvous. Although the use of these or equivalent methods is necessary to protect the consistency of shared data or to guarantee the proper use of non-preemptable resources, their use may jeopardize the ability of the system to meet its timing requirements. In fact, a direct application of these synchronization mechanisms may lead to an indefinite period of priority inversion and low schedulability. However, the discussion is limited to scheduling within a uniprocessor. Readers who are interested in the multiprocessor synchronization problem are referred to (Rajkumar, Sha and Lehockzy 1988).

Example 1: Suppose J_1, J_2, and J_3 are three jobs arranged in descending order of priority with J_1 having the highest priority. Let jobs J_1 and J_2 share a data structure guarded by a binary semaphore S. Suppose that at time t_1, job J_3 locks the semaphore S and executes its critical section. During the execution of job J_3's critical section, the high priority job J_1 is initiated, preempts J_3 and later attempts to use the shared data. However, job J_1 will be blocked on the semaphore S. One might expect that J_1, being the highest priority job, is blocked no longer than the time for job J_3 to complete its critical section. However, the duration of blocking is, in fact, unpredictable. This is because job J_3 can be preempted by the intermediate priority job J_2. The blocking of J_3, and hence that of J_1, will continue until J_2 and any other pending intermediate jobs are completed.

The blocking period in this example can be arbitrarily long. One way to deal with the priority inversion problem is to let the critical section of each task to run to completion without interruption. This is know as the *kernelized monitor approach* (Mok 1983), which is an effective approach for short critical sections. Another approach is to properly manage task interactions. The *priority ceiling protocol* is a scheme designed for the use of binary semaphores. This protocol ensures (1) freedom from mutual deadlock and (2) that a high priority task will be blocked by lower priority tasks for the duration of at most one critical section (Goodenough and Sha 1988; Sha, Rajkumar and Lehoczky 1987).

Two ideas underlie the design of this protocol. First is the concept of priority inheritance: when a task τ blocks the execution of higher priority tasks, task τ should execute at the highest priority level of all the tasks blocked by τ. Secondly, we must guarantee that each newly started critical section executes at a priority level that is higher than the (inherited) priority levels of the preempted critical sections. It was shown in (Sha, Rajkumar, and Lehoczky 1987) that such a prioritized total ordering in the execution of critical sections leads to the two properties mentioned above. To achieve such prioritized total ordering, we define the concept of the priority ceiling of a binary semaphore S to be equal to the highest priority task that may lock S. When a job J attempts to execute one of its critical sections, it will be blocked unless its priority is strictly higher than all the priority ceilings of semaphores currently locked by jobs other than J. If job J blocks, the job that holds the lock on the highest priority ceiling semaphore is said to be blocking J and hence inherits J's priority. A job J can, however, always preempt another job executing at a lower priority level as long as J does not attempt to enter a critical section.

Example 2: Suppose that we have two jobs J_1 and J_2 in the system. In addition, there are two shared data structures protected by binary semaphores S_1 and S_2 respectively. Suppose the sequence of processing steps for each job is as follows:

$$J_1 = \{\ldots, \mathbf{P}(S_1), \ldots, \mathbf{P}(S2), \ldots, \mathbf{V}(S_2), \ldots, \mathbf{V}(S_1), \ldots\}$$

$$J_2 = \{\ldots, \mathbf{P}(S_2), \ldots, \mathbf{P}(S_1), \ldots, \mathbf{V}(S_1), \ldots\ \mathbf{V}(S_2), \ldots\}\ .$$

Recall that the priority of job J_1 is assumed to be higher than that of job J_2. Thus, the priority ceilings of both semaphores S_1 and S_2 are equal to the priority of job J_1. Suppose that at time t_0, J_2 is initiated and it begins execution and then locks semaphore S_2. At time t_1, job J_1 is initiated and preempts job J_2 and at time t_2, job J_1 tries to enter its critical section by making an indivisible system call to execute $\mathbf{P}(S_1)$. However, the runtime system will find that job J_1's priority is *not* higher than the priority ceiling of *locked* semaphore S_2. Hence, the runtime system suspends job J_1 without locking S_1. Job J_2 now *inherits* the priority of job J_1 and resumes execution. Note that J_1 is blocked outside its critical section. As J_1 is not given the lock on S_1 but suspended instead, the potential deadlock involving J_1 and J_2 is prevented. Once J_2 exits its critical section, it will return to its assigned priority and immediately be preempted by job J_1. The J_1 will execute to completion, and finally J_2 will resume and run to completion.

Let B_i be the longest duration of blocking that can be experienced by a job of task τ_i. The following two theorems indicate whether the deadlines of a set of periodic tasks can be met if the priority ceiling protocol is used.

THEOREM 3: A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm if the following condition is satisfied (Sha, Rajkumar and Lehoczky 1987):

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} + max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{1/n} - 1)\ .$$

THEOREM 4: A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm for all task phasings if the following condition is satisfied (Sha, Rajkumar and Lehoczky 1987).

$$\forall i,\ 1 \le i \le n,\ \min_{(k,l)\epsilon R_i} \left(\sum_{j=1}^{i-1} C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \right) \le 1$$

where C_i, T_i and R_i are defined in Theorem 2, and B_i is the worst-case blocking time for a job of task τ_ι.

Remark: Theorems 3 and 4 generalize Theorems 1 and 2 by taking the blocking duration of a job into consideration. The B_i's in Theorems 3 and 4 can be used to account for any delay caused by resource sharing. Note that the upper limit of the summation in the theorem is $(i - 1)$ instead of i, as in Theorem 2.

In the application of Theorems 3 and 4, it is important to realize that under the priority ceiling protocol, a task τ can be blocked by a lower priority task τ_L if τ_L may lock a semaphore S whose priority ceiling is higher than or equal to the priority of task τ, even if τ and τ_L do not share any semaphore. For example, suppose that τ_L locks S first. Next, τ is initiated and preempts τ_L. Later, a high priority task τ_H is initiated and attempts to lock S. Task τ_H will be blocked. Task τ_L now *inherits* the priority of τ_H and executes. Note that τ has to wait for the critical section of τ_L even τ and τ_L do not share any semaphore. We call such blocking, ***push-through blocking***. Push-through blocking is the price for avoiding unbounded priority inversion. If task τ_L does not inherit the priority of τ_H, task τ_H can be indirectly preempted by task τ and all the tasks that have priority higher than that of τ_L. Finally, we want to point out that even if task τ_H does not attempt to lock S but attempts to lock another unlocked semaphore, τ_H will still be blocked by the priority ceiling protocol because τ_H's priority is not higher than the priority ceiling of S. We call this form of blocking, ***ceiling blocking***. Ceiling blocking is the price for ensuring the freedom of deadlock and the property of a task being blocked at most once. Both ceiling blocking and push-through are accounted for by B_i in Theorems 3 and 4.

2.3. An example

In this section, we give a simple example to illustrate the application of the scheduling theorems.

Example 3: We would like to check the schedulability of the following task set.

1. Periodic task τ_1: execution = 40 msec; period = 100 msec; deadline is at the end of each period.

   In addition, τ_3 may block τ_1 for 10 msec through the use of a shared communication server and task τ_2 may block τ_1 for 20 msec through the use of a shared data object.
2. Periodic task τ_2: execution time = 40 msec; period = 150 msec; deadline is 20 msec before the end of each period.
3. Periodic task τ_3: execution time = 100 msec; period = 350 msec; deadline is at the end of each period.

Since under the priority ceiling protocol a task can be blocked by lower priority tasks at most once, the maximal blocking time for task τ_1 is $B_1 = \max(10, 20)$ msec $= 20$ msec. Since τ_3 may lock the semaphore S_c associated with the communication server and the priority ceiling of S_c is higher than that of task τ_2, task τ_2 can be blocked by task τ_3 for 10 msec.[2] Finally, task τ_2 has to finish 20 msec earlier than the nominal deadline of a periodic task. This is equivalent to saying that τ_2 will always be blocked for additional 20 msec but its deadline is at the end of the period. Hence, $B_2 = (10 + 20)$ msec $= 30$ msec.[3] Using Theorem 4:

1. Task τ_1: Check $C_1 + B_1 \leq 100$. Since $40 + 20 \leq 100$, task τ_1 is schedulable.
2. Task τ_2: Check whether either

$$\begin{array}{lll} & C_1 + C_2 + B_2 \leq 100 & 40 + 40 + 30 > 100 \\ \text{or} & 2C_1 + C_2 + B_2 \leq 150 & 80 + 40 + 30 = 150 \end{array}$$

   Task τ_2 is schedulable and in the worst-case phasing will meet its deadline exactly at time 150.
3. Task τ_3: Check whether either

$$\begin{array}{lll} & C_1 + C_2 + C_3 \leq 100 & 40 + 40 + 100 > 100 \\ \text{or} & 2C_1 + C_2 + C_3 \leq 150 & 80 + 40 + 100 > 150 \\ \text{or} & 2C_1 + 2C_2 + C_3 \leq 200 & 80 + 80 + 100 > 200 \\ \text{or} & 3C_1 + 2C_2 + C_3 \leq 300 & 120 + 80 + 100 = 300 \\ \text{or} & 4C_1 + 3C_2 - C_3 \leq 350 & 160 + 120 + 100 > 350 \end{array}$$

Task τ_3 is also schedulable and in the worst-case phasing will meet its deadline exactly at time 300. If follows that all the three periodic tasks can meet their deadlines.

3. Mode change protocols

We now discuss the protocols needed to support mode changes in the context of our scheduling algorithms for periodic tasks. First, we discuss the characteristics of mode change. This is followed by a simple protocol when only independent tasks are involved. Finally, we discuss the mode change problems in the presence of task interactions.

3.1. Mode changes for independent tasks

From a scheduling point of view, typical mode change operations can be classified into two types:

1. Operations that increase a task set's processor utilization:
   a. Adding a task
   b. Increasing the execution time of a task
   c. Increasing the frequency of execution of a task.

2. Operations that decrease a task set's processor utilization:
   a. Deleting a task
   b. Decreasing the execution time of a task
   c. Decreasing the frequency of a task.

A simple mode change protocol can be defined in terms of the deletion of existing tasks and the addition of a new task. If a task modifies its parameters, that is, changes its sampling rate, it is modeled as the deletion of the original task and the addition of a new task. In addition, we assume that all the tasks are periodic and that a task which has started its execution will not be deleted until it has completed its execution in the current period. We will relax these assumptions later in this paper, however.

When tasks are independent, the addition, deletion, or modification of a task's parameters is merely an application of Theorems 1 or 2.

THEOREM 5: At any time t, a task τ can be added, or its computation time C increased or its frequency increased without causing any task to miss their deadlines if the conditions of Theorems 1 or 2 are satisfied.

Proof: It directly follows from the fact that a task set is schedulable if it satisfies the conditions of Theorems 1 or 2.

THEOREM 6: At any time t, a task τ can be deleted, or its computation time C reduced or its frequency reduced without causing any task to miss their deadlines.

Proof: It directly follows the fact that if a given task set satisfies the conditions of Theorems 1 or 2, then the modified task set will also satisfy the conditions in question.

It may seem that once a task is deleted, its allocated processor capacity can be immediately reused by other tasks. However, this is not true. The schedulability of a set of tasks using the rate-monotonic algorithm is determined under the assumption that once a job J of a task τ is initiated, task τ cannot request additional processing until the beginning of τ's next period. Thus, even if job J has finished its execution m units before the end of τ's current period, task τ has used up the processor capacity for the given period. Hence, task τ must be included in the application of Theorems 1 and 2, 3 and 4 until the end of the current period. In other words, the processor capacity allocated to τ cannot be used by new tasks until the end of τ's current period.

3.2. The basic mode change protocol

In this section, we will develop a basic mode change protocol for periodic tasks using binary semaphores for synchronization. There are two basic concepts in the design of this protocol. The first is the notion of sufficient processor capacity to add a task *on the fly* when synchronization is involved. The second is the preservation of the characteristic of the priority ceiling protocol: each newly started critical section is guaranteed to execute at a priority level that is higher than the maximum priority that any of the preempted critical sections can inherit.

Definition: Processor capacity is said to be sufficient for adding a task τ, if the resulting n tasks, including τ, can meet all their deadlines using the rate-monotonic algorithm and the priority ceiling protocol.

Theorems 3 and 4 provide us with sufficient conditions for processor capacity to be sufficient. Theorem 4 allows for a higher degree of processor utilization while Theorem 3 is easier to apply.

We have defined the concept of having sufficient capacity to add a task. A related concept is the deletion of a task τ and reclaiming the processor capacity used by τ.

Definition: The processor capacity used by a deleted task τ is said to be reclaimed at time t if after t task τ does not need to be included in the application of Theorems 3 and 4.

We now define the basic mode change protocol. We assume that during mode transition, tasks are deleted/added in an order that is consistent with the semantics of the application.

1. The addition and/or the deletion of tasks in mode change may lead to the modification of the priority ceiling of some semaphores across the mode change. Upon the initiation of mode change,
   - For each of the unlocked semaphores S, whose priority ceiling needs to be raised, S's ceiling is raised immediately and indivisibly.
   - For each locked semaphore S, whose priority ceiling needs to be raised, S's priority ceiling is raised immediately and indivisibly after S is unlocked.
   - For each semaphore S, whose priority ceiling needs to be lowered, S's priority ceiling is lowered when all the tasks which may lock S and which have priorities greater than the new priority ceiling of S are deleted.
2. A task τ, which needs to be deleted, can be deleted immediately upon the initiation of mode change, if τ has not yet started its execution in its current period. In addition, the processor capacity used by τ is reclaimed immediately. On the other hand, if τ has started execution, τ can be deleted after the end of its execution and before its next initiation time. The processor capacity allocated to τ will, however, not be reclaimed until the next initiation time.
3. A task τ can be added into the system if the following two conditions are met:
   - If task τ's priority is higher than the priority ceiling of locked semaphores $S_1, \ldots,$ S_k, then the priority ceiling of $S_1, \ldots S_k$ must be first raised before adding task τ.
   - There must be sufficient processor capacity for adding task τ.

We now illustrate the mode change protocol using an example.

Example 4: Suppose that the task set $\{\tau_1, \tau_2, \tau_2\}$ is replaced by the task set $\{\tau_0, \tau_3^*,$ $\tau_4, \tau_5\}$. In other words, tasks τ_1 and τ_2 are to be deleted and replaced with τ_0, τ_4 and τ_5 in the new task set. The task τ_3 is to be modified to τ_3^*resulting in a change of parameters. Suppose that τ_0 cannot be added until τ_1 is deleted because of insufficient processor capacity or semantic requirements. Similarly, suppose that τ_4 and τ_5 cannot be added until τ_2 is deleted and its processor capacity reclaimed. We assume that we add tasks τ_0, τ_4 and τ_5

in that order when a mode change is initiated. In addition, we assume that tasks that need to be deleted can be deleted in any order.

Let the jobs of each task execute the following sequences of instructions in the current task set.

$$\begin{aligned}
J_1 &= \{\ldots, \mathbf{P}(S_1), \ldots, \mathbf{V}(S_1), \ldots\}\\
J_2 &= \{\ldots, \mathbf{P}(S_1), \ldots, \mathbf{P}(S_2), \ldots, \mathbf{V}(S_2), \ldots, \mathbf{V}(S_1), \ldots\}\\
J_3 &= \{\ldots, \mathbf{P}(S_2), \ldots, \mathbf{V}(S_2), \ldots\}\,.
\end{aligned}$$

Let the jobs in the new mode execute the following sequences of events:

$$\begin{aligned}
J_0 &= \{\ldots, \mathbf{P}(S_2), \ldots, \mathbf{V}(S_2), \ldots\}\\
J_3^* &= \{\ldots, \mathbf{P}(S_2), \ldots, \mathbf{V}(S_2), \ldots\}\\
J_4 &= \{\ldots, \mathbf{P}(S_2), \ldots, \mathbf{P}(S_1), \ldots, \mathbf{V}(S_1), \ldots, \mathbf{V}(S_2), \ldots\}\\
J_5 &= \{\ldots, \mathbf{P}(S_1), \ldots, \mathbf{V}(S_1), \ldots\}\,.
\end{aligned}$$

As before, we assume that the priority of J_{i+1} is lower than the priority of J_i. Before the mode change, the priority ceilings of S_1 and S_2 are the priorities of τ_1 and τ_2 respectively. However, after the mode change, the priority ceilings of S_1 and S_2 are the priorities of τ_4 and τ_0 respectively. Thus, after the mode change, the priority ceiling of S_1 is lowered, while that of S_2 is raised.

Consider the following sequence of events depicted in Figure 1. A line at a low level indicates that the corresponding job is blocked or has been preempted by a higher priority job. A line raised to a higher level indicates that the job is executing. The absence of a line indicates that the job has not yet been initiated or has completed. Shaded portions indicate execution of critical sections.

- At time t_0, the task set that is being run is $\{\tau_1, \tau_2, \tau_3\}$. J_3 arrives and begins execution.
- At time t_1, J_3 locks S_2 and enters its critical section.
- At time t_2, J_2 arrives and preempts J_3.
- At time t_3, J_2 attempts to lock S_1 and is blocked by the priority ceiling protocol. J_3 inherits J_2's priority and resumes execution.
- At time t_4, J_1 arrives and preempts J_3.
- At time t_5, J_1 successfully locks S_1, since its priority is higher than the priority ceiling of locked semaphore S_2.
- At time t_6, J_1 releases the semaphore S_1. At the same time, a mode change is initiated due to external requirements. τ_0 is the first task to be added at the mode change and it cannot be added until the processing capacity is reclaimed from τ_1. Hence, τ_0 cannot be added until the end of J_1's current period (at t_{12}). Similarly, τ_4 and τ_5 cannot be added until the end of J_2's current period (at t_{17}). The priority ceiling of S_2 gets raised in the new mode but cannot be raised until it is unlocked.
- At time t_7, J_1 completes execution and J_3 resumes execution at its inherited priority of J_2.
- At time t_8, J_3 releases the semaphore S_2 and resumes its original priority. The priority ceiling of S_2 is raised now. J_2 immediately preempts J_3 and locks S_1.
- At time t_9, J_2 makes a nested access to S_2 and locks S_2.

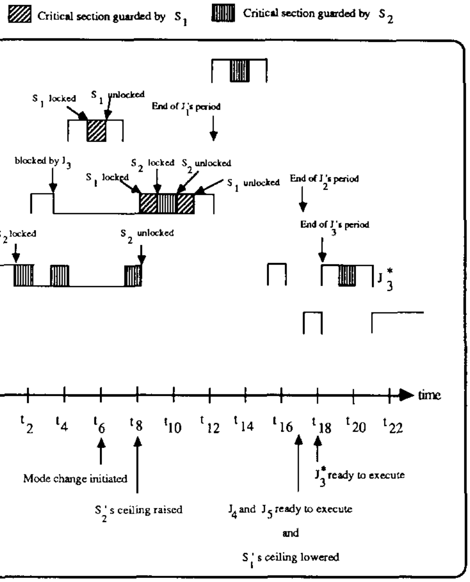

Figure 1. Sequence of events described in example 4.

- At time t_{10}, J_2 releases the semaphore S_2.
- At time t_{11}, J_2 releases the semaphore S_1.
- At time t_{12}, J_2 completes execution. The current period τ_1 ends here and τ_1 is deleted. Hence, τ_0 is added into the system and immediately becomes eligible for execution.
- At time t_{13}, J_0 locks semaphore S_2 since there is no other locked semaphore in the system.
- At time t_{14}, J_0 releases S_2.
- At time t_{15}, J_0 completes execution and J_3 resumes execution.
- At time t_{16}, J_3 completes execution.
- The processor remains idle during the interval $[t_{16}, t_{17}]$.[4]
- At time t_{17}, the current period of τ_2 ends and it can be deleted from the system. The priority ceiling of S_1 is lowered and τ_0 was added into the system. Now, τ_4 and τ_5 can also be added into the system. Having the highest priority among tasks ready to run, J_4 begins execution.
- At time t_{18}, τ_3's current period ends and τ_3 can be replaced with τ_3^*. The mode change is now complete. Job J_3 preempts J_4 and begins execution.
- At time t_{21}, J_3^* completes execution, locking and releasing S_2 at t_{19} and t_{20} respectively. Now, J_4 resumes execution.
- Processing proceeds normally in the new mode.

The above example illustrates the following properties of the mode change protocol. First, tasks can be added as long as they are schedulable in the resulting task set. However, a task to be added may have to wait for the deletion of an existing task even though there is idle capacity available. We shall further study this mode change delay in Section 3.4. Task modifications, such as the modification of τ_3 into τ_3^* can be carried out relatively easily.

3.3. Properties of the basic mode change protocol

The Mode Change Protocol is designed to keep the properties of the priority ceiling protocol valid. Under the priority ceiling protocol, there is no mutual deadlock, and a job can be blocked by lower priority jobs for at most the duration of a single critical section (Sha, Rajkumar and Lehoczky 1987). We shall now prove that both these properties are preserved under the mode change protocol.

Lemma 7: Under the mode change protocol, when a job J enters its critical section and preempts job J_i while J_i is in its critical section, the priority of J is higher than the priority that can be inherited by J_i.

Proof: Under the definition of the mode change protocol, the priority ceiling of a semaphore S will not be lower than the priority of any job that may lock S. When job J enters its critical section, its priority will be higher than the (inherited) priority of the jobs preempted by J, since (1) a job J is allowed to enter its critical section only if J's priority is higher than the priority ceilings of all the semaphores locked by jobs other than J, and (2) the highest priority that a job can inherit is bounded by the priority ceiling of the semaphores locked by this job.

THEOREM 8: There is no mutual deadlock under the mode change protocol.

Proof: Suppose that there is a mutual deadlock. Let the highest priority of all the jobs involved in the deadlock be P. Due to the transitivity of priority inheritance, all the jobs involved in the deadlock will eventually inherit the same highest priority P. This contradicts Lemma 7.

Lemma 9: A job J can be blocked by a lower priority job J_L at most for the duration of executing one critical section.

Proof: First, if job J_L is not already in its critical section when job J arrives, then job J_L will be preempted by J and cannot block J. Suppose that J_L is in its critical section when J arrives and that J_L blocks J. J_L inherits the priority of J and continues its execution. Once J_L exits its critical section, by the definition of the priority ceiling protocol, J_L will be assigned its original priority and be immediately preempted by J. Hence, J_L cannot block J again.

THEOREM 10: Under the mode change protocol, a job J can be blocked by lower priority jobs for at most the duration of a single (outermost) critical section.

Proof: Suppose that job J is blocked by lower priority jobs more than once. By Lemma 9, job J must be blocked by n different lower priority jobs, $J_1, \ldots, J_n$, where the priority of J_i is assumed to be higher than or equal to that of J_{i+1}. Since a lower priority job cannot block a higher priority job unless it is already in its critical section, jobs $J_1, \ldots, J_n$ must be in their critical sections when J arrives. By assumption, J is blocked by J_n and J_n

inherits the priority of J. It follows that job J's priority cannot be higher than the highest priority P that can be inherited by J_n. On the other hand, by Lemma 7, job J_{n-1}'s priority is higher than P. It follows that job J_{n-1}'s priority is higher than that of job J. This contradicts the assumption that J's priority is higher than that of jobs $J_1, \ldots, J_n$.

Remark: It is important to point out that the property of a job being blocked for at most one critical section depends upon our model of a job, an instance of a periodic task. We assume that when a job executes alone on the processor, input data from I/O devices will be ready when the job is initiated and it will continue to execute until it completes without suspension for I/O activities. In some applications, an instance of a periodic task may need to suspend itself for I/O. In this case, we have the following corollary:

Corollary 11: If a generalized job J suspends itself n times during its execution, it can be blocked for the duration of at most $n + 1$ critical sections.

3.4. Mode change delays.

In this section, we analyze the delays that can occur before a mode change is completed. In the following analysis, we assume that the given task set is schedulable using the rate-monotonic algorithm and the priority ceiling protocol.

Notation: Let t_0 denote the time at which the mode change is initiated. Let D_s be the delay in elevating semaphore priority ceilings, that is, the delay between t_0 and the time at which all the semaphores whose ceilings need to be raised are raised. Let D_c be the delay in reclaiming processor capacity, that is, the delay between t_0 and the time at which all the tasks that need to be deleted are deleted and their allocated processor capacity becomes available. Finally, let D be the *mode change delay,* that is, the duration between t_0 and the time at which the mode change is completed.

The following lemmas and theorems are based on the assumption that the task sets before and after a mode change are schedulable.

Lemma 12: Let S_i be a semaphore whose priority ceiling needs to be raised during a mode change. Let τ be a task whose priority is equal to the priority ceiling of semaphore S_i. The delay in elevating the priority ceiling of S_i is bounded by the period of task τ, T.

Proof: The priority ceiling of a semaphore S can be raised only if it is not locked. Semaphore S may have been locked when the mode change is initiated. However, under the assumption that task τ can meet its deadline, the locking of S cannot be longer than T and the lemma follows.

Lemma 13: Let S^* be the semaphore that has the lowest priority ceiling of all the semaphores whose ceilings need to be raised. The ceiling elevation delay for mode change, D_s is bounded by T^*, the period of a task whose priority is equal to the priority ceiling of S^*.

Proof: It directly follows from Lemma 12 and from the fact that a task associated with a lower priority ceiling has a longer period under the rate-monotonic scheduling algorithm.

Lemma 14: Let task τ be the lowest priority task needed to be deleted. The delay due to the reclamation of processor capacity, D_b, is bounded by the period of task τ.

Proof: Let the periods of the tasks that need to be deleted be $\{T_j, \ldots, T_m\}$, where $T_k \geq T_j$ if $k > j$. Under the assumption that the set of given tasks is schedulable, each of the tasks needed to be deleted can be deleted by the end of its current period, and its allocated processor capacity can be reclaimed. Hence, we have $D_c = max\{T_j, \ldots, T_m\}$. Under the rate-monotonic scheduling algorithm, a task with a longer period has lower priority. It follows that the delay due to reclaiming all the processor capacity is bounded by the period of task τ_m, T_m.

THEOREM 15: The mode change delay D is bounded by $max(D_s, D_c)$.

Proof: Suppose that the mode change request occurs at time t_0. By $(t_0 + D_s)$, all the semaphore priority ceilings that need to be raised have been raised. By $(t_0 + D_c)$, all the tasks in the current mode that need to be deleted are deleted. That is, the processor capacity needed for the new tasks is available by $(t_0 + D_c)$. It follows that all the new tasks can be added by the time $(t_0 + max(D_s, D_c))$. Finally, by the definition of the basic mode change protocol, all the semaphore priority ceilings that need to be lowered have been lowered by $(t_0 + D_c)$. Hence, the mode change delay is bounded by $max(D_s, D_c)$.

Remark: Under the mode change protocol, the maximal mode change delay is bounded by the longest period in a task set, which is generally much shorter and will never be longer than the least common multiple (LCM) of all the periods. In the cyclical executive approach, the major cycle is the LCM of all the periods and a mode change will not be initiated until the current major cycle completes. Hence, the delay to complete a mode change using the mode change protocol would typically be much shorter than the delay using the cyclical executive approach. In addition, the mode change protocol also provides the flexibility of adding the most urgent task in the new mode first.

4. Extensions of the basic mode change protocol

In this section, we will examine some design alternatives to the mode change protocol as well as the integration of our basic mode change protocol with other scheduling algorithms.

4.1. Variations of the basic protocol

The objective in the design of the mode change protocol is to minimize the mode change delay subject to keeping the shared data consistent and to meeting all the deadlines of tasks that must be continuously executing. We also made an implicit assumption that the mode change protocol should not lower the system schedulability in any given mode.

However, assumptions and objectives are, of course, application dependent. Generally, there is relatively little that one can do about mode change delay caused by reclaiming processor capacity, because a task could have started its execution when the mode change is initiated. Once a task begins execution, it may well be desirable to let it complete because the abortion of a task may lead to complications that makes later correction and/or recovery time-consuming. There is an exception to this general observation, however. In certain applications, one can define a set of tasks that constitutes an atomic configuration unit. Such a unit encapsulates all the shared variables for the task set in question. In this case, the application semantic may allow the entire unit to be deleted immediately and indivisibly at the initiation of mode change.

Generally, when $D_s > D_c$, there is an incentive to minimize D_s. We can minimize the mode change delay associated with elevating the priority ceilings if we are willing to pay a schedulability cost. For example, we define the *global ceiling* mode change protocol as follows. In this protocol, the priority ceiling of a semaphore S is defined as the priority of the highest priority task that may access S across *all* modes. The disadvantage of this mode change protocol is rather obvious. In any mode, the *actual* ceiling of a semaphore can be much lower than the *global* priority ceiling. As a consequence, the blocking duration is longer and it translates into schedulability cost, causing some otherwise schedulable task sets to become unschedulable. The priority ceiling elevation cost can be fine-tuned, however. Since D_s is determined solely by the period of the task whose priority equals the lowest priority ceiling that needs to be raised, D_s can be shortened by deliberately assigning a higher priority ceiling to the semaphore with this lowest priority ceiling that needs to be raised in mode changes.

Finally, we may want to emphasize the simplicity of managing a mode change process. In this case, we do not raise the semaphore priority ceiling of any semaphore until all the tasks that need to be deleted are deleted and the priority ceilings of associated semaphores are lowered. New tasks will be added at time $t_{add} = t_0 + (D_s + D_c)$. We need apply neither Theorem 3 nor Theorem 4 during runtime as long as tasks are known to be schedulable in each mode. This is because at time t_{add} all the deleted tasks' processor capacity have already been reclaimed and the priority ceilings are at the correct level. That is, the condition under which we may apply Theorem 3 or 4 is the same as in the new mode.

4.2. Stability under transient overload

In this section, we discuss the integration between the mode change protocol and the solution to the stability problem. In the previous sections, the computation time of a task is assumed to be constant. However, in many applications, task execution times are often stochastic, and the worst-case execution time can be significantly larger than the average execution time. In order to have a reasonably high average processor utilization, we must deal with the problem of transient overload. We consider a scheduling algorithm to be *stable* if there exists a set of *critical* tasks such that all tasks in the set will meet their deadlines even if the processor is overloaded. This means that under worst-case conditions, tasks outside the critical set may miss their deadlines. The rate-monotonic algorithm is stable in the sense that the set of tasks that never miss their deadlines does not change as the processor gets more overloaded or as task phasings change. Of course, which tasks are in the critical task set depends on the worst-case utilizations of the particular tasks being considered. The important point is that the rate-monotonic theory guarantees that if such a set exists, it always consists of tasks with the highest priorities. This means that if a transient overload should develop, tasks with longer periods will miss their deadlines.

Of course, a task with a longer period could be more critical to an application than a task with a shorter period. One might attempt to ensure that the critical task always meets its deadline by assigning priorities according to a task's importance. However, this approach can lead to poor schedulability, that is, with this approach, deadlines of critical tasks might be met only when the total utilization is low.

The *period transformation* technique can be used to ensure high utilization while meeting the deadline of an important, long-period task. Period transformation means turning a long-period important task into a high priority task by splitting its work over several short periods. For example, suppose task τ with a long period T is not in the critical task set and must never miss its deadline. We can make τ simulate a short period task by giving it a period of $T/2$ and suspending it after it executes half its worst-case execution time, $C/2$. The task is then resumed and finishes its work in the next execution period. It still completes its total computation before the end of period T. From the viewpoint of the rate-monotonic theory, the transformed task has the same utilization but a shorter period, $T/2$, and its priority is raised accordingly. It is important to note that the most important task does not have to have the shortest period. We only need to make sure that it is among the first n high priority tasks whose worst-case utilization is within the scheduling bound. A systematic procedure for period transformation with minimal task partitioning can be found in (Sha, Lehoczky and Rajkumar 1986).

Period transformation allows important tasks to have higher priority while keeping priority assignments consistent with rate-monotonic rules. This kind of transformation should be familiar to users of cyclic executives. The difference here is that we don't need to adjust the code segment sizes so different code segments fit into shared time slots. Instead, τ simply requests suspension after performing $C/2$ amount of work. Alternatively, the runtime scheduler can be instructed to suspend the task after a certain amount of computation has been done, without affecting the application code.[5]

The period transformation approach has another benefit—it can raise the rate-monotonic utilization bound. Suppose the rate-monotonic utilization bound is $U_{max} < 100$ percent, that is, total task utilization cannot be increased above U_{max} without missing a deadline. When a period transformation is applied to the task set, U_{max} will rise. For example:

Example 4: Let

- Task τ_1: $C_1 = 4$; $T_1 = 10$; $U_1 = .400$
- Task τ_2: $C_2 = 6$; $T_2 = 14$; $U_1 = .428$.

The total utilization is .828, which just equals the bound of Theorem 1, so this set of two tasks is schedulable. If we apply Theorem 2, we find:

$$C_1 + C_2 \leq T_1 \qquad 4 + 6 = 10 \qquad l = 1, k = 1$$

or $$2C_1 + C_2 \leq T_2 \qquad 8 + 6 = 14 \qquad l = 1, k = 2.$$

So Theorem 2 says the task set is just schedulable. Now suppose we perform a period transformation on task τ_1, so $C_1' = 2$ and $T_1' = 5$. The total utilization is the same and the set is still schedulable, but when we apply Theorem 2 we find:

$$\begin{array}{llll} & C_1 + C_2 \leq T_1 & 2 + 6 > 5 & l = 1, k = 1 \\ \text{or} & 2C_1 + C_2 \leq 2T_1 & 4 + 6 = 10 & l = 2, k = 1 \\ \text{or} & 3C_1 + C_2 < T_2 & 6 + 6 < 14 & l = 1, k = 2. \end{array}$$

The third inequality shows that the compute times for tasks τ_1 and/or τ_2 can be increased without violating the constraint. For example: the compute time of Task τ_1 can be increased by 2/3 units to 2.667, giving an overall schedulable utilization of 2.667/5 + 6/14 = .961; or the compute time of Task τ_2 can be increased to 8, giving an overall schedulable utilization of 2/5 + 8/14 = .971. So the effect of the period transformation has been to raise the utilization bound from .828 to at least .961 and at most .971. Indeed, if periods are uniformly harmonic, that is, if each period is an integral multiple of each shorter period, the utilization bound of the rate-monotonic algorithm is 100 percent.[6] So the utilization produced by the rate-monotonic approach is only an upper bound on what can be achieved if the periods are not transformed. Of course, as the periods get shorter, the scheduling overhead utilization increases, so the amount of useful work that can be done decreases. For example, before a period transformation, the utilization for a task, including scheduling overhead, is $(C + 2S)/T$, where $2S$ is the context switching time due to the preemption and resumption of a task. After splitting the period into two parts, the utilization is $(.5C + 2S)/.5T$, so scheduling overhead is a larger part of the total utilization. However, the utilization bound is also increased, in general. If the increase in utilization caused by the scheduling overhead is less than the increase in the utilization bound, then the period transformation is a win—more useful work can be done while meeting all deadlines.

Period transformation does not affect the mode change protocol except that to delete a transformed task that has already started execution, we must wait for its completion which may take several *transformed periods*. In addition, we cannot reclaim the processor capacity of a transformed task until the end of the last transformed period, which is also the end of the task's original period.

4.3. Scheduling both aperiodic and periodic tasks

It is important to meet the regular deadlines of periodic tasks *and* the response time requirements of aperiodic events. We now review the scheduling of both aperiodic and periodic tasks within the rate monotonic framework.[7] As we will see, the mode change protocol can easily accommodate the aperiodic scheduling algorithms. Let us begin with a simple example.

Suppose that we have two tasks. Let τ_1 be a periodic task with period 100 and execution time 99. Let τ_2 be an aperiodic task that appears once within a period of 100 but the arrival time is random. The execution time of task τ_2 is one unit. If we let the aperiodic task wait for the periodic task, then the average response time is about 50 units. The same can be said for a polling server, which provides one unit of service time in a period of 100. On the other hand, we can deposit one unit of service time in a *ticket box* every 100 units of time; when a new ticket is deposited, the unused old tickets, if any, are discarded. With this approach, no matter when the aperiodic event arrives during a period of 100, it will find there is a ticket for one unit of execution time at the ticket-box. That is, τ_2 can use the ticket to preempt τ_1 and execute immediately when the event occurs. In this case, τ_2's response time is precisely one unit and the deadlines of τ_1 are still guaranteed. This is the idea behind the *deferrable* server algorithm (Lehoczky, Sha and Strosnider 1987), which reduces aperiodic response time by a factor of about 50 in this example.

In reality, there can be many periodic tasks whose periods can be arbitrary. Furthermore, aperiodic arrivals can be very bursty, as for a Poisson process. However, the idea remains unchanged. We should allow the aperiodic tasks to preempt the periodic tasks subject to not causing their deadlines to be missed. It was shown in (Lehoczky, Sha and Strosnider 1987) that the deadlines of periodic tasks can be guaranteed provided that during a period of T_a units of time, there are no more than C_a units of time in which aperiodic tasks preempt periodic tasks. In addition, the total periodic and aperiodic utilization must be kept below $(U_a + ln[(2 + U_a)/(2U_a + 1))]$, where $U_a = C_a/T_a$. And the server's period must observe the inequality $T_a \le (T - C_a)$, where T is the period of a periodic task whose priority is just lower than that of the server.

Compared with background service, the deferrable server algorithm typically improves aperiodic response time by a factor between 2 and 10 (Lehoczky, Sha and Strosnider 1987). Under the deferrable server algorithm, both periodic and aperiodic task modules can be modified at will as long as the utilization bound is observed.

A variation of the deferrable server algorithm is known as the *sporadic* server algorithm (Sprunt, Sha and Lehoczky 1987). As for the deferrable server algorithm, we allocate C_a units of computation time within a period of T_a units of time. However, the C_a of the server's budget is not refreshed until the budget is consumed.[8] From a capacity planning point of view, a sporadic server is equivalent to a periodic task that performs polling. That is, we can place sporadic servers at various priority levels and use only Theorems 1 and 2 to perform a schedulability analysis. Sporadic and deferrable servers have similar performance gains over polling because any time an aperiodic task arrives, it can use the allocated budget immediately. When polling is used, however, an aperiodic arrival generally needs to wait for the next instant of polling. The sporadic server has the least runtime overhead. Both the polling and the deferrable servers have to be serviced periodically, even if there are no aperiodic arrivals.[9] There is no overhead for the sporadic server until its execution budget has been consumed. In particular, there is no overhead if there are no aperiodic arrivals. Therefore, the sporadic server is especially suitable for handling emergency aperiodic events that occur rarely but must be responded to quickly.

Simulation studies of the sporadic server algorithm (Sprunt, Sha and Lehoczky 1989) show that given a lightly loaded server, aperiodic events are served 5–10 times faster than with background service, and 3–6 times faster than with polling. Figure 2, from (Sprunt, Sha and Lehoczky 1989), shows one example of the relative performance between background execution, the deferrable server algorithm (DS), the sporadic server algorithm (SS), polling, and another algorithm, not explained here, called the priority exchange algorithm (PE). The analysis underlying these results assumes a Poisson arrival process with exponentially distributed service time. In addition, each server (other than the background server)

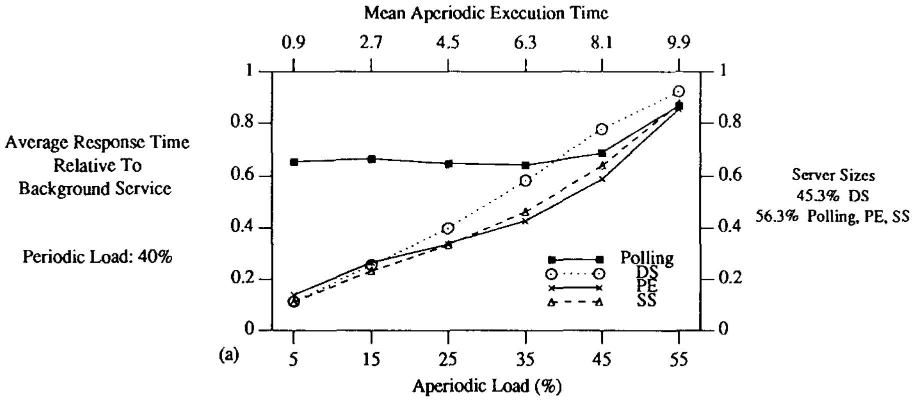

Figure 2. Scheduling both aperiodic and periodic tasks.

is given a period that allows it to execute as the highest priority task.[10] Aperiodic requests can therefore preempt the execution of periodic tasks as long as server execution time is available.

The maximum amount of aperiodic service time allowed before periodic tasks will miss their deadline is called the *maximum server size*. In this example, aperiodic tasks can preempt periodic tasks for at most 56.3 percent of the sporadic or polling server's period without causing the deadlines of periodic tasks to be missed. For the deferrable server, only a smaller amount of service time is possible: 43.6 percent. In either case, the server is not allowed to execute at its assigned priority once its computation budget is exhausted, although it can continue to execute at background priority if time is available. A server's budget is refreshed at the end of its period, at which time execution can resume at the server's assigned priority.

Figure 2 shows the average response times of the different scheduling algorithms as a function of average aperiodic workload. When the average aperiodic workload is small compared with the sporadic server size, randomly arriving requests are likely to find the server available and can successfully preempt the periodic tasks. This results in good performance. For example, when the average aperiodic workload is 5 percent,[11] the deferrable and sporadic server response time is about 10 percent of the average background response time, while the average polling response time is about 65 percent of background response time. (This means the sporadic server gives about 6 times faster response than polling and 10 times faster than background service.) When the aperiodic workload increases, the likelihood of server availability decreases and the resulting performance advantage also decreases. For example, when the aperiodic load is 55 percent, the different server algorithms do not give significant performance improvement over background service.

From a mode change point of view, sporadic server effectively transforms the service of aperiodic events into periodic tasks. We can add or delete a server and increase or decrease its capacity as if it were a normal periodic task.

5. Conclusions

In many real-time applications, neither the task set nor the task priorities remain static throughout the mission. A change in operational mode often leads to the modification of task parameters as well as the addition of new tasks and deletion of old tasks. In this paper, we have developed a simple mode change protocol in a prioritized preemptive scheduling environment. We have shown that under this mode change protocol, there cannot be mutual deadlocks, and a high priority job can be blocked by lower priority jobs for at most the duration of one critical section, despite the addition and deletion of tasks during the mode change. We have shown that the worst-case mode change delay under this protocol is bounded and is generally much shorter than that possible in a commonly used cyclical executive.

Acknowledgement

The authors wish to thank John Goodenough for his helpful comments.

Notes

1. It was shown in (Liu and Layland 1973) that when all the tasks are initiated at the same time, if the first job of a task meets its deadline, that task will never miss a deadline.
2. This may occur if τ_3 blocks τ_1 and inherits τ_1's priority.
3. Note that the blocked-at-most-once result does not apply here. It only applies to blocking caused by task synchronization using the priority ceiling protocol.
4. Idling of the processor can occur for two reasons: the rate-monotonic algorithm does not guarantee a 100 percent schedulability level for all task sets. Secondly, task sets in some modes may have lower processor utilization levels than task sets in other modes.
5. The scheduler must ensure that τ is not suspended while in a critical region since such a suspension can cause other tasks to miss their deadlines. If the suspension time arrives but the task is in a critical region, then the suspension should be delayed until the task exits the critical region. To account for this effect on the schedulability of the task set, the worst-case execution time must be increased by ϵ, the extra time spent in the critical region, that is, τ's utilization becomes $(0.5C + \epsilon)/0.5T$.
6. For example, by transforming the periods in Example 3 so τ_1' and τ_2' both have periods of 50, the utilization bound is 100 percent, that is, 4.7 percent more work can be done without missing a deadline.
7. Aperiodic tasks are used to service aperiodic events.
8. Early refreshing is also possible under certain conditions. See (Sprunt, Sha and Lehoczky 1989).
9. The ticket box must be refreshed at the end of each deferrable server's period.
10. This means each server's period must not be greater than the shortest period of all the periodic tasks. The sporadic server and polling server can have a period equal to that of the shortest period task. As mentioned earlier in this section, however, the deferrable server must have an even shorter period.
11. A 5 percent average aperiodic workload means that in the long run, the aperiodic requests consume about 5 percent of the CPU cycles, although the number of requests and their execution time vary from period to period and from request to request.

References

Goodenough, J. B., and L. Sha 1988. The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks. *The Proceedings of the 2nd ACM International Workshop on Real-Time Ada Issues.*

Lehoczky, J. P. and L. Sha 1986. Performance of Real-Time Bus Scheduling Algorithms. *ACM Performance Evaluation Review, Special Issue.* 14, (1) (May).

Lehoczky, J. P., L. Sha, and J. Strosnider 1987. Enhancing Aperiodic Responsiveness in a Hard Real-Time Environment. *IEEE Real-Time System Symposium.*

Lehoczky, J. P., L. Sha, and Y. Ding 1987. *The Rate Monotonic Scheduling Algorithm—Characterization and Average Case Behavior.* Technical Report, Department of Statistics, Carnegie Mellon University.

Leinbaugh, D. W. 1980. Guaranteed Response Time in a Hard Real-Time Environment. *IEEE Transactions on Software Engineering,* (Jan).

Leung, J. Y. and M. L. Merrill 1980. A Note on Preemptive Scheduling of Periodic Real-Time Tasks. *Information Processing Letters* 11 (3) (Nov.): 115–118.

Liu, C. L. and J. W. Layland 1973. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *JACM* 20 (1): 46–61.

Mok, A. K. 1983. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* Ph.D. thesis, Massachusetts Institute of Technology.

Rajkumar, R., L. Sha, and L. Lehoczky 1987. On Countering the Effect of Cycle Stealing in A Hard Real-Time Environment. *IEEE Real-Time System Symposium.*

Rajkumar, R., L. Sha, and J. P. Lehockzy 1988. Real-Time Synchronization Protocols for Multiprocessors. *Proceedings of the IEEE Real-Time Systems Symposium.*

Ramaritham, K. and J. A. Stankovic 1984. Dynamic Task Scheduling in Hard Real-Time Distributed Systems. *IEEE Software* (July).

Sha, L., J. P. Lohoczky, and R. Rajkumar 1986. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. *IEEE Real-Time Systems Symposium.*

Sha, L., R. Rajkumar, and J. P. Lehoczky 1987. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical Report, Department of Computer Science, Carnegie Mellon University (To appear in *IEEE Transactions on Computers*).

Sprunt, B., L. Sha, and J. P. Lehoczky 1989. Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System. *Real-Time Systems* 1 (1) (June).

Zhao, W., K. Ramamritham, and J. Stankovic 1987. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers,* (Aug).