Technical Report

CMU/SEI-88-TR-033
ESD-TR-88-034

Real-Time Scheduling Theory and Ada

Lui Sha
John B. Goodenough

November 1988

Technical Report

CMU/SEI-88-TR-033
ESD-TR-88-034
November 1988

Real-Time Scheduling Theory and Ada

Lui Sha
John B. Goodenough

Real-Time Scheduling in Ada Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Real-Time Scheduling Theory and Ada

Abstract: The Ada tasking model was intended to facilitate the management of concurrency in a priority-driven scheduling environment. In this paper, we will review some important results of a priority-based scheduling theory, illustrate its applications with examples, discuss its implications for the Ada tasking model, and suggest workarounds that permit us to implement analytical scheduling algorithms within the existing framework of Ada.

1. Introduction

1.1. Background

Traditionally, many real-time systems use cyclical executives to schedule concurrent threads of execution. Under this approach, a programmer lays out the execution timeline by hand to serialize the execution of critical sections and to meet task deadlines. While such an approach is adequate for simple systems, it quickly becomes unmanageable for large systems. It is a painful process to iteratively divide high level language code so the compiled *machine code* segments can be fitted into time slots of a cyclical executive and that critical sections of different tasks do not interleave. Forcing programmers to schedule tasks by fitting machine code slices on a timeline is no better than the outdated approach of managing memory by manual memory overlay. Such an approach often destroys program structure and results in real-time programs that are difficult to understand and maintain.

The Ada tasking model represents a fundamental departure from the cyclical executive model. Indeed, the dynamic preemption of tasks at runtime generates non-deterministic timelines that are at odds with the very idea of the fixed execution timeline required by a cyclical executive. From the viewpoint of real-time scheduling theory, an Ada task represents a concurrent unit for scheduling. As long as the real-time scheduling algorithms are supported by the Ada runtime and the resource utilization bounds on CPU, I/O drivers, and communication media are observed, the timing constraints will be guaranteed. Even if there is a transient overload, a fixed subset of *critical* tasks can still meet their deadlines as long as they are schedulable by themselves. In other words, the integration of Ada tasking with analytical scheduling algorithms allows programmers to meet timing constraints by managing resource requirements and relative task importance. This makes Ada tasking truly useful for real-time applications while also making real-time systems easier to develop and maintain.

1.2. Controlling Priority Inversion

To put real-time scheduling on an analytical basis, systems must be built in a way that ensures that high-priority tasks are minimally delayed by lower priority tasks when both are contending for the same resources. *Priority inversion* occurs when the use of a resource by a low priority task delays the execution of a high priority task. Priority inversion occurs either when task priorities are incorrectly assigned or when they are not used correctly when allocating resources. One common mistake in priority scheduling is assigning priorities solely according to task importance.

Example 1: Suppose that τ_1 and τ_2 are periodic tasks with periods 100 and 10, respectively. Both of them are initiated at $t = 0$, and task τ_1 is more important than task τ_2. Assume that task τ_1 requires 10 units of execution time and its first deadline is at $t = 100$, while task τ_2 needs 1 unit of execution time with its first deadline at $t = 10$. If task τ_1 is assigned higher scheduling priority because of its importance, task τ_2 will miss its deadline unnecessarily even though the total processor utilization is only 0.2. Both tasks can meet their deadlines using the rate monotonic algorithm [5], which assigns higher priorities to tasks with shorter periods. In fact, many more new tasks can be added into the system by using the simple rate monotonic scheduling algorithm.

Although priority inversion is undesirable, it cannot be completely eliminated. For example, when a low priority task is in a critical region, the higher priority task that needs the shared data must wait. Nonetheless, the duration of priority inversion must be tightly bounded in order to ensure a high degree of responsiveness and schedulability. Controlling priority inversion is a system level problem. The tasking model, runtime support, program design, and hardware architecture should all be part of the solution, not part of the problem. For example, there is a serious priority inversion problem in some existing IEEE 802.5 token ring implementations. While there are 8 priority levels in the token arbitration, the queueing of message packets is FIFO, i.e., message priorities are ignored. As a result, when a high priority packet is behind a low priority packet, the high priority packet has to wait not only for the lower priority packet to be transmitted, but also for the transmission of all medium priority packets in the network. The result is "hurry-up at the processor but miss the deadline at the communication network." Using FIFO queueing in a real-time system is a classical case of priority inversion and can lead to extremely poor schedulability. Priority assignments must be observed at every level of a system for all forms of resource allocation. Minimizing the duration of priority inversion is the key to meeting deadlines and keeping systems responsive to aperiodic events.

Section 2 reviews some of the important results in real-time scheduling theory. We begin with the problem of scheduling independent periodic tasks. Next, we address the issues of maintaining stability under transient overload and the problem of scheduling both periodic and aperiodic tasks. We conclude Section 2 by reviewing the problems of real-time synchronization. In Section 3, we review the Ada tasking scheduling model and suggest some workarounds that permit us to implement many of the scheduling algorithms within the framework of existing Ada rules. Finally, we conclude this paper in Section 4.

2. Scheduling Real-Time Tasks

In this section, we provide an overview of some of the important issues of a real-time scheduling theory. We will begin with the problem of ensuring that independent periodic tasks meet their deadlines. Next, we show how to ensure that critical tasks meet their deadlines even when a system is temporarily overloaded. We then address the problem of scheduling both periodic and aperiodic tasks. Finally, we conclude this section by reviewing the problems of real-time synchronization and communication.

2.1. Periodic Tasks

Tasks are *independent* if their executions need not be synchronized. Given a set of independent periodic tasks, the *rate monotonic scheduling algorithm* gives each task a fixed priority and assigns higher priorities to tasks with shorter periods. A task set is said to be *schedulable* if all its deadlines are met, i.e., 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 [5].

> **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
>
> $$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \le n(2^{1/n} - 1) = U(n)$$
>
> 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% ($ln\ 2$) as the number of tasks approaches infinity. Table 2-1 shows values of the bound for one to nine tasks.

U(1) = 1.0	U(4) = 0.756	U(7) = 0.728
U(2) = 0.828	U(5) = 0.743	U(8) = 0.724
U(3) = 0.779	U(6) = 0.734	U(9) = 0.720

Table 2-1: Scheduling Bounds for One to Nine Independent Tasks

The bound of Theorem 1 is very pessimistic because the worst-case task set is contrived and unlikely to be encountered in practice. For a randomly chosen task set, the likely bound is 88% [3]. 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 must be checked [3].

> **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

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

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

This theorem provides the exact schedulability criterion for independent periodic task sets under 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 ends of periods of higher priority tasks within τ's first deadline. In the formula, i denotes the task to be checked and k denotes each of the tasks that affects the completion time of task i, i.e., task i and the higher priority tasks. For a given i and k, each value of l represents the scheduling points of task 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, i.e., $i = k = 1$ and ($l = 1, \cdots, \lfloor T_i/T_k \rfloor = \lfloor T_1/T_1 \rfloor = 1$). The scheduling point is, of course, τ_1's first deadline ($l\, T_k = 5, l = 1, k = 1$). For task τ_i ($i = 2$), there are *two* scheduling points from all higher priority tasks, τ_k ($k = 1$), i.e., ($l = 1, \cdots, \lfloor T_i/T_k \rfloor = \lfloor T_2/T_1 \rfloor = 2$). The two scheduling points are, of course, the two end points of task τ_1's period within the first deadline of task τ_2 at 14, i.e., ($l\, T_k = 5, l = 1, k = 1$) and ($l\, T_k = 10, l = 2, k = 1$). Finally, there is the scheduling point from τ_2's own first deadline, i.e., ($l\, T_k = 14, l = 1, k = 2$). At each scheduling point, we check if the task in question can complete its execution at or before the scheduling point. This is illustrated in detail by Examples 3 and 8 below.

Example 2: Consider the case of three periodic tasks, where $U_i = C_i/T_i$.

- Task τ_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0.2$
- Task τ_2: $C_2 = 40$; $T_2 = 150$; $U_2 = 0.267$
- Task τ_3: $C_3 = 100$; $T_3 = 350$; $U_3 = 0.286$

The total utilization of these three tasks is 0.753, which is below Theorem 1's bound for three tasks: $3(2^{1/3} - 1) = 0.779$. Hence, we know these three tasks are schedulable, i.e., they will meet their deadlines if τ_1 is given the highest priority, τ_2 the next highest, and τ_3 the lowest.

The remaining 24.7% processor capacity can be used for low priority background processing. However, we can also use it for additional hard real-time computation.

Example 3: Suppose we replace τ_1's algorithm with one that is more accurate and computationally intensive. Suppose the new algorithm doubles τ_1's computation time from 20 to 40, so the total processor utilization increases from 0.753 to 0.953. Since the utilization of the

[1]It was shown in [5] that when all the tasks are initiated at the same time (the worst-case phasing), if a task completes its execution before the end of its first period, it will never miss a deadline.

first two tasks is 0.667, which is below Theorem 1's bound for two tasks, $2(2^{1/2} - 1) = 0.828$, the first two tasks cannot miss their deadlines. For task τ_3, we use Theorem 2 to check whether the task set is schedulable, i.e., we set $i = n = 3$, and check whether one of the following equations holds:

$$\forall\, k, l,\ 1 \le k,l \le 3, \quad \sum_{j=1}^{3} \left\lceil \frac{l\, T_k}{T_j} \right\rceil C_j \le l\, T_k$$

To check if task τ_3 can meet its deadline, it is only necessary to check the equation for values of l and k such that $l\, T_k \le T_3 = 350$. If one of the equations is satisfied, the task set is schedulable.

	$C_1 + C_2 + C_3 \le T_1$	$40 + 40 + 100 > 100$	$l = 1, k = 1$
or	$2C_1 + C_2 + C_3 \le T_2$	$80 + 40 + 100 > 150$	$l = 1, k = 2$
or	$2C_1 + 2C_2 + C_3 \le 2T_1$	$80 + 80 + 100 > 200$	$l = 2, k = 1$
or	$3C_1 + 2C_2 + C_3 \le 2T_2$	$120 + 80 + 100 = 300$	$l = 2, k = 2$, or $l = 3, k = 1$[2]
or	$4C_1 + 3C_2 + C_3 \le T_3$	$160 + 120 + 100 > 350$	$l = 1, k = 3$

The analysis shows that task τ_3 is also schedulable and in the worst-case phasing will meet its deadline exactly at time 300. Hence, we can double the utilization of the first task from 20% to 40% and still meet all the deadlines. The remaining 4.7% processor capacity can be used for either background processing or a fourth hard deadline task, which has a period longer than that of τ_3[3] and which satisfies the condition of Theorem 2.

A major advantage of using the rate monotonic algorithm is that it allows us to separate logical correctness concerns from timing correctness concerns. Suppose that a cyclical executive is used for this example. The major cycle must be the least common multiple of the task periods. In this example, the task periods are in the ratio 100:150:350 = 2:3:7. A minor cycle of 50 units would induce a major cycle of 42 minor cycles, which is an overly complex design. To reduce the number of minor cycles, we can try to modify the periods. For example, it might be possible to reduce the period of the longest task, from 350 to 300. The total utilization is then exactly 100%, and the period ratios are 2:3:6; the major cycle can then be 6 minor cycles of 50 units. To implement this approach and minimize the splitting of computations belonging to a single task, we could split task τ_1 into two parts of 20 units computation each, $C_{1,1}$ and $C_{1,2}$. The computation of task τ_2 similarly could be split into at least two parts such that task τ_3 need only be split into four parts. A possible timeline in-

[2]That is, after 300 units of time, τ_1 will have run three times, τ_2 will have run twice, and τ_3 will have run once. The required amount of computation just fits within the allowed time, so each task meets its deadline. [5] showed that since the tasks meet their deadlines at least once within the period T_3, they will always meet their deadlines.

[3]Task τ_3 just meets its deadline at 300 and hence we cannot add a task with a priority higher than that of task τ_3.

dicating the amount of computation for each task in each minor cycle is shown in the following table, where 20_1 on the first line indicates the first part of task τ_1's computation, which takes 20 units of time.

Cyclic Timeline for Example 3						
	1	2	3	4	5	6
τ_1	20_1	20_2	20_1	20_2	20_1	20_2
τ_2	30_1	10_2		30_1	10_2	
τ_3		20_1	30_2		20_3	30_4

Table 2-2: Minor Cycle Timeline: Each minor cycle is 50.

When processor utilization level is high and there are many tasks, fitting code segments into time slots can be a time-consuming iterative process. In addition, later modification of any task may overflow a particular minor cycle and require the entire timeline to be redone. But more important, the cyclic executive approach has required us to modify the period of one of the tasks, increasing the utilization to 100% *without in fact doing more useful work.* Under the rate monotonic approach, all deadlines are met, but machine utilization is 95.3% instead of 100%. This doesn't mean the rate monotonic approach is less efficient. The capacity that isn't needed to service real-time tasks can be used by background tasks, e.g., for built-in-test purposes. With the cyclic executive approach, no such additional work can be done in this example. Of course, the scheduling overhead for preemptive scheduling needs to be taken into account. If S is the amount of time needed for a single scheduling action, then since there are two scheduling actions per task, the total utilization devoted to scheduling is $2S/T_1 + 2S/T_2 + 2S/T_3$. In the cyclic case, the scheduling overhead is partly in the time needed to dispatch each task's code segment in each minor cycle and partly in the utilization wasted by decreasing the period for task 3. For this example, the scheduling overhead for the cyclic approach is at least 4.7%:

Actual_Utilization – Required_Utilization, i.e.,
100/300 – 100/350 = .333 – .286 = .047

Thus, although the rate monotonic approach may seem to yield a lower maximum utilization than the cyclic approach, in practice, the cyclic approach may simply be consuming more machine time because the periods have been artificially shortened. In addition, cyclic executives get complicated when they have to deal with aperiodic events. The rate monotonic approach, as will be discussed later, readily accommodates aperiodic processing. Finally, the rate monotonic utilization bound as computed by Theorem 2 is a function of the periods and computation times of the task set. The utilization bound can always be increased by transforming task periods, as described in the next section.

2.2. Stability Under Transient Overload

In the previous section, 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. 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 need not 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 [7].

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.[4]

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%, i.e., 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:

<u>Example 4:</u> Let

- Task τ_1: $C_1 = 4$; $T_1 = 10$; $U_1 = .400$
- Task τ_2: $C_2 = 6$; $T_2 = 14$; $U_2 = .428$

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

	$C_1 + C_2 \le T_1$	$4 + 6 = 10$	$l = 1, k = 1$
or	$2C_1 + C_2 \le T_2$	$8 + 6 = 14$	$l = 1, k = 2$

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:

	$C_1 + C_2 \le T_1$	$2 + 6 > 5$	$l = 1, k = 1$
or	$2C_1 + C_2 \le 2T_1$	$4 + 6 = 10$	$l = 2, k = 1$
or	$3C_1 + C_2 < T_2$	$6 + 6 < 14$	$l = 1, k = 2$

The third equation 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, i.e., if each period is an integral multiple of each shorter period, the utilization bound of the rate monotonic algorithm is 100%.[5] So the utilization bound 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 in-

[4]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, i.e., τ's utilization becomes $(0.5C+\varepsilon)/0.5T$.

[5]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%, i.e., 4.7% more work can be done without missing a deadline.

creases, 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 $(C + 2S)/T$. 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.

2.3. 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. ("Aperiodic tasks" are used to service such events.) Let us begin with a simple example.

Example 5: 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 [4], 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 [4] 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 "$T_a \leq (T - C_a)$", where T is the period of a periodic task whose priority is next to the server.

Compared with background service, the deferrable server algorithm typically improves aperiodic response time by a factor between 2 and 10 [4]. Under the deferrable server algorithm, both periodic and aperiodic task modules can be modified at will as long as the utilization bound is observed. Figure 2-1 illustrates the relative performance between background execution, the deferrable server algorithm, and polling. The workload is 60% periodic and 20% aperiodic. We assume a Poisson arrival process and exponentially distributed execution time for the aperiodic tasks. Since the mean aperiodic workload is fixed at 20%, short mean interarrival times imply short mean execution times. As we can see from Figure 2-1, the deferrable server is most effective for frequent arrivals with small service times.

Figure 2-1: Scheduling Both Aperiodic and Periodic Tasks

A variation to the deferrable server algorithm is known as the *sporadic* server algorithm [9]. 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.[6] 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.[7] 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.

[6]Early refreshing is also possible under certain conditions. See [9].

[7]The ticket box must be refreshed at the end of each deferrable server's period.

2.4. Task Synchronization

In the previous sections we have discussed the scheduling of independent tasks. Tasks, however, do interact. In this section, we will discuss how the rate monotonic scheduling theory can be applied to real-time tasks that must interact. The discussion is limited in this paper to scheduling within a uniprocessor. Readers who are interested in the multiprocessor synchronization problem should see [6].

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.

Example 6: Suppose J_1, J_2, and J_3 are three jobs arranged in descending order of priority with J_1 having the highest priority. We assume that jobs J_1 and J_3 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 the critical section of job J_3, 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. We would hope 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. This situation can be partially remedied if a job in its critical section is not allowed to be preempted; however, this solution is only appropriate for very short critical sections because it creates unnecessary blocking. For instance, once a low priority job enters a long critical section, a high priority job that does not access the shared data structure may be needlessly blocked.

The *priority ceiling protocol* is a real-time synchronization protocol with two important properties: 1) freedom from mutual deadlock, and 2) bounded blocking, i.e., at most one lower priority task can block a higher priority task [1, 8]. There are two ideas in 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 executes at the highest priority level of all the tasks blocked by τ. Secondly, we must guarantee that a critical section is allowed to start execution only if the section will always execute at a priority level that is higher than the (inherited) priority levels of any preempted critical sections. It was shown in [8] that such a prioritized total ordering in the execution of critical sections leads to the two desired properties. To achieve such prioritized total ordering, we define the *priority ceiling* of a binary semaphore S to be the highest priority of all tasks that may lock S. When a task τ attempts to execute one of its critical sections, it will be suspended unless its priority is higher than the priority ceilings of all semaphores currently locked by tasks other than τ. If task τ is unable to enter its critical section for this reason, the task that holds the lock on the semaphore with the highest priority ceiling is said to be blocking τ and hence inherits the priority of τ. As long as a task τ is not attempting to enter one of its critical sections, it will preempt every task that has a lower priority.

Example 7: 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.

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

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

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 the priority of J_1 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. From this point on, J_1 will execute to completion, and then J_2 will resume its execution until its completion.

Let B_i be the longest duration of blocking that can be experienced by 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, for all task phasings, if the following condition is satisfied [8]:

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

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 [8].

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

where C_i, T_i, and R_i are defined in Theorem 2, and B_i is the worst-case blocking time for τ_i.

Remark: Theorems 3 and 4 generalize Theorems 1 and 2 by taking blocking 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 though 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 the priority of τ_H is not higher than the priority ceiling of *S*. We term this form of blocking as *ceiling blocking*. Ceiling block is the price for ensuring the freedom of deadlock and the property of a task being blocked at most once.

2.5. An Example Application of the Theory

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

Example 8: Consider the following task set.

1. Emergency handling task: execution time = 5 msec; worst case interarrival time = 50 msec; deadline is 6 msec after arrival.
2. Aperiodic event handling tasks: average execution time = 2 msec; average inter-arrival time = 40 msec; fast response time is desirable but there are no hard deadlines.
3. Periodic task τ_1: execution time = 20 msec; period = 100 msec; deadline is at the end of each period.

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

Solution: First, we create a sporadic server for the emergency task, with a period of 50 msec and a service time 5 msec. Since the server has the shortest period, the rate monotonic algorithm will give this server the highest priority. It follows that the emergency task can meet its deadline.

Since the aperiodic tasks have no deadlines, they can be assigned a low background priority. However, since fast response time is desirable, we create a sporadic server executing at the second highest priority. The size of the server is a design issue. A larger server (i.e., a server with higher utilization) needs more processor cycles but will give better response time. In this example, we choose a large server with a period of 100 msec and a service time of 10 msec. We now have two tasks with a period of 100 msec, the aperiodic server and periodic task τ_1. The rate monotonic algorithm allows us to break the tie arbitrarily, and we let the server have the higher priority.

We now have to check if the three periodic tasks can meet their deadlines. 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.[8] Finally, task τ_2 has to finish 20 msec earlier than the nominal deadline for a periodic task. This is equivalent to saying that τ_2 will always be blocked for an additional 20 msec but its deadline is at the end of the period. Hence, B_2 = (10 + 20) msec = 30 msec.[9] At this point, we can directly apply Theorem 4. However, we can also reduce the number of steps in the analysis by noting that period 50 and 100 are harmonics and we can treat the emergency server as if it has a period of 100 msec and a service time of 10 msec, instead of a period of 50 msec and a service time of 5 msec. We now have three tasks with a period of 100 msec and an execution time of 20 msec, 10 msec, and 10 msec respectively. For the purpose of analysis, these three tasks can be replaced by a single periodic task with a period of 100 msec and an execution time of 40 msec (20 + 10 + 10). We now have the following three equivalent periodic tasks for analysis:

- Task τ_1: $C_1 = 40$; $T_1 = 100$; $B_1 = 20$; $U_1 = 0.4$
- Task τ_2: $C_2 = 40$; $T_2 = 150$; $B_2 = 30$; $U_2 = 0.267$
- Task τ_3: $C_3 = 100$; $T_3 = 350$; $B_3 = 0$; $U_3 = 0.286$

Using Theorem 4:

1. Task τ_1: Check $C_1 + B_1 \le T_1$. Since $40 + 20 \le 100$, task τ_1 is schedulable.
2. Task τ_2: Check whether either

	$C_1 + C_2 + B_2 \le T_1$	$40 + 40 + 30 > 100$
or	$2C_1 + C_2 + B_2 \le T_2$	$80 + 40 + 30 = 150$

   Task τ_2 is schedulable and in the worst-case phasing will meet its deadline exactly at time 150.

[8]This may occur if τ_3 blocks τ_1 and inherits the priority of τ_1.

[9]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.

3. Task τ_3: Check whether either

	$C_1 + C_2 + C_3 \leq T_1$	$40 + 40 + 100 > 100$
or	$2C_1 + C_2 + C_3 \leq T_2$	$80 + 40 + 100 > 150$
or	$2C_1 + 2C_2 + C_3 \leq 2T_1$	$80 + 80 + 100 > 200$
or	$3C_1 + 2C_2 + C_3 \leq 2T_2$	$120 + 80 + 100 = 300$
or	$4C_1 + 3C_2 + C_3 \leq T_3$	$160 + 120 + 100 > 350$

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

We now determine the response time of the aperiodics. The server capacity is 10% and the average aperiodic workload is 5% (2/40). Because most of the aperiodic arrivals can find "tickets," we would expect a good response time. Indeed, using a M/M/1 [2] approximation for the lightly loaded server, the expected response time for the aperiodics is $W = E[S]/(1 - \rho) = 2/(1 - (0.05/0.10)) = 4$ msec, where E[S] is the average execution time of aperiodic tasks and ρ is the average server utilization. Finally, we want to point out that although the worst-case total periodic and server workload is 95%, we can still do quite a bit of background processing since the soft deadline aperiodics and the emergency task are unlikely to fully utilize the servers.

3. Real-Time Scheduling in Ada

The Real-Time Scheduling in Ada Project at the Software Engineering Institute is a cooperative effort between the SEI, system developers in industry, Ada vendors, and DoD agencies. It aims at applying the scheduling theory reviewed in Section 2 to the design and implementation of hard real-time systems in Ada. The deferred server, sporadic server, and priority ceiling scheduling algorithms have all been developed at Carnegie Mellon University. We call these *analytic* scheduling algorithms because the overall timing behavior of a system is subject to mathematical analysis, starting with the basic rate monotonic theorems. The project is implementing these analytic scheduling algorithms using an Ada runtime system, and is coding examples of real-time systems to evaluate the suitability of the whole approach.

While the use of Ada for real-time systems presents many practical, implementation-dependent questions such as the cost of performing a rendezvous, there are other important questions about the suitability of the concepts and scheduling rules of Ada. For example, tasks in Ada run non-deterministically, making it hard for traditional real-time programmers to decide whether any tasks will meet their deadlines. In addition, the scheduling rules of Ada don't seem to support prioritized scheduling well. Prioritized tasks are queued in FIFO order rather than by priority, high priority tasks can be delayed indefinitely when calling low priority tasks, and task priorities cannot be changed when application demands change at runtime. Fortunately, it appears that none of these problems present insurmountable difficulties; solutions exist within the current language framework, although some language changes would be helpful. In the SEI Real-Time Scheduling in Ada Project, we are investigating how to apply analyzable scheduling algorithms within the Ada scheduling rules; we are also investigating how the scheduling rules of Ada might be changed to make Ada more directly suitable for real-time work. In addition, we are formulating Ada coding guidelines to use when coding hard real-time systems. These guidelines allow the project's scheduling algorithms to be applied and supported within the existing language rules. The guidelines are still evolving and being evaluated, but so far, it seems likely they will meet the needs of a useful range of systems.

3.1. On Ada Scheduling Rules

First of all, the Ada tasking model is well-suited, in principle, to the use of real-time scheduling algorithms. When using this approach, a programmer doesn't need to know when each task is running to be sure that deadlines will be met. That is, both Ada and the theory abstract away the details of an execution timeline and view tasks as the basic unit of abstraction for the management of concurrency. Although Ada tasks fit well with the theory at the conceptual level, Ada and the theory differ on the rules for determining when a task is eligible to run and its execution priority. For example, if a high priority task calls a lower priority task that is in rendezvous with another low priority task, the rendezvous continues at the priority of the task being served instead of being increased because a high priority task is waiting. Under these circumstances, the high priority task can be blocked as long as there are medium priority jobs able to run. But there are a variety of solutions to this problem. The most general solution within the constraints of the language is simply to not use pragma PRIORITY at all. If all tasks in a system have no assigned priority, then the scheduler is free to use any convenient algorithm for deciding which eligible task to run. An implementation-dependent pragma could be used to give "scheduling priorities" to tasks, i.e., indications of scheduling importance that would be used in accordance with analytic scheduling algorithms. This approach would even allow "priorities" to be changed dynamically by the programmer because such changes only affect the scheduling of tasks that, in a legalistic sense, have no Ada priorities at all. The only problem with this approach is that tasks are still queued in FIFO order rather than by priority. However, this problem can often be solved by using a coding style that prevents queues from having more than one task, making the FIFO issue irrelevant. Of course, telling programmers to assign "scheduling priorities" to tasks but not to use pragma PRIORITY, and being careful to avoid queueing tasks surely says we are fighting the language rather than taking advantage of it.

While this kind of drastic approach may be necessary in some cases, it nonetheless seems likely that many real-time systems can be programmed satisfactorily using the concept of priorities in Ada and giving an appropriate interpretation to Ada scheduling rules. The relevant Ada rules are:

- Task priorities: fixed. This rule is inappropriate when task priorities need to be changed at runtime. For example, when a new mode is initiated, the frequency of a task and/or its criticality may change, implying its priority must change. In addition, the priority of a sporadic server for aperiodic events needs to be lowered when "tickets" are used up, and needs to be raised when new "tickets" arrive.

  Solution: When an application needs to adjust the priority of a task at runtime, this task should be declared as having no Ada priority. The runtime system can then be given a way of adjusting the priority of the task.

- CPU allocation: priorities must be observed. Ada requires that the highest priority task eligible to run be given the CPU when this is "sensible." "Sensible" for a uniprocessor is usually interpreted to mean that if a call by an executing task can be accepted, the call *must* be accepted and no lower priority task can be allowed to execute. Although this interpretation may seem to be obviously the best, it is in fact not correct for the priority ceiling protocol, which gives better service to high priority tasks by ensuring that they are blocked at most once by lower priority tasks. For example (see Figure 3-1), suppose a critical region is implemented as a server task S, whose accept statements are the critical regions guarded by a semaphore, and suppose a low priority task is in rendezvous with S. Also suppose the priority ceiling of S is H (i.e., the highest priority task that can call S has priority H). Now suppose execution of the rendezvous is preempted by a medium priority task M, whose priority is less than H. Suppose M tries to call T, a server other than S. The priority ceiling protocol says that the call from M must be blocked, but the normal interpretation of Ada's scheduling rules would imply that the call of M must be accepted since M is the highest priority task that is eligible to run and T is able to accept the call.

  Solution: M can be suspended just as it is about to call T because the priority ceiling protocol says it isn't "sensible" to let server T execute on behalf of M. So M is suspended, and the execution of S continues to the end of the rendezvous. At that point, the priority ceiling protocol allows the call from M to succeed.

Alternatively, give S and T an undefined priority. That is, the Ada rules say S has no defined priority outside a rendezvous and within a rendezvous; it has "at least" the priority of the calling task. These rules give an implementation sufficient freedom to support the priority ceiling protocol directly. For example, S is allowed to preempt M just at the point where M is about to call T.

Figure 3-1: Blocking Due to the Priority Ceiling Protocol

- Hardware task priority: always higher than software task priorities. This Ada rule reflects current hardware designs, but hardware interrupts should not always have the highest priority, from the viewpoint of the rate monotonic theory.

  Solution: When handling an interrupt that, in terms of the rate monotonic theory, should have a lower priority than the priority of some application task, keep the interrupt handling actions short (which is already a common practice) and include the interrupt handling duration as blocking time in the rate monotonic analysis.
- Priority rules for task rendezvous:
  - Selective wait: priority can be ignored. That is, the scheduler is allowed, but not required, to take priorities into account.

    Solution: Ensure the runtime system takes priorities of waiting tasks into account.
  - FIFO entry queues: the priority of calling tasks must be ignored.

    Solution: Often, it is possible to prevent queues from being formed (as explained in the next section). If there are no queues, no issue of FIFO queueing arises. If it is impossible to prevent the formation of an entry queue, entry families can be used to get the effect of prioritized queueing.
  - Called task priority: only increased during rendezvous.

Solution: Don't specify the priority of the called task. If the called task has no assigned priority, its effective priority can be increased according to priority inheritance or other scheduling rules. Alternatively, give the called task a priority just higher than the priority of any of its callers. From an analytical point of view, this causes no additional blocking in the worst case, i.e., this approach does not reduce the schedulability of the system. However, when the server is called by a low priority task, it can now preempt some tasks that would otherwise be executable under the priority ceiling protocol. This can cause the average response time for aperiodic servers to be worse, and can increase the probability that a non-critical task will miss its deadline in a transient overload situation. But if the service time for low priority tasks is short, these negative effects will be negligible, so this is often a viable approach that can be used with today's Ada runtime systems.

From what our project has learned so far, it seems to be possible in practice to support analytic scheduling algorithms under current Ada rules by using a combination of runtime system modifications and appropriate coding protocols. Of course, it would be better if the language did not get in the way of priority scheduling principles. The future revision of Ada should probably revise some of these rules so priority-based scheduling can be supported more directly within the language.

3.2. Ada Real-Time Design Guidelines

While different guidelines can be applied to different real-time programming problems, the guidelines developed in the SEI project reflect a basic principle of real-time programming — write systems in a way that minimizes the duration of priority inversion. In other words, minimize the time a high priority task has to wait for the execution of lower priority tasks.

For example, consider a set of periodic tasks that must exchange data among themselves or that must call a communication server. The periodic tasks are called *client* tasks. They do not call each other. Whenever they must read or write shared data or send a message, they call a *server* task whose entries represent the critical regions. The client tasks are given priorities according to rate monotonic principles, i.e., tasks with the shortest periods are given the highest priorities. There are two options when assigning a priority to a server. If the Ada runtime system supports the priority ceiling protocol directly, then give the server a low priority or an undefined priority. In addition, tell the runtime system the priority ceiling of the server, i.e., the highest priority of all its clients. The server will then, in effect, execute at its client's priority when there are no other callers. When a high priority client wants to call a server, the call may be blocked in accordance with priority ceiling rules because these rules specify when it is "sensible" to allow the high priority task to continue to execute.

If the priority ceiling protocol is not supported directly by the runtime system, the priority ceiling protocol can be approximated by assigning each server task a unique priority that is one greater than the priority of its highest priority caller. This approach can be used today with all Ada implementations. Theoretically, assigning the server the same priority as its highest priority caller is sufficient for the approximation. However, different Ada implementations treat tasks of equal priority differently. We have found that some implementations are inconsistent with the treatment demanded by the priority ceiling protocol for equal priority tasks, namely, they do not give preference to the server. Increasing the priority by one avoids this problem as long as there is no client task at this priority level.

With either approach, as long as a server need not synchronize with external events, it will be either waiting for a client or serving its first caller at a priority that prevents other clients from calling the server. As a result, the server task never has more than one client on its queues, so prioritized queueing is not a problem. However, when a server needs to synchronize with external I/O events, it is suspended while in a rendezvous. Other client tasks can then execute and attempt to call the server, and FIFO queues can develop. To ensure that entry queues do not develop, the runtime system should suspend tasks that attempt to call a server and allow some lower priority task to run. This has the same effect as a prioritized queue because the suspended tasks will be awakened in priority order; the highest priority calling task will have its call accepted first. If this solution is not supported by the runtime system, then entry families must be used to simulate priority queues.

From a theoretical viewpoint, the time spent in each rendezvous with a server is counted as part of the computing time, C_i, for the client task τ_i. Since the use of servers is a synchronization problem, Theorems 3 and 4 apply. Under the priority ceiling protocol, the maximal blocking time for a client task at priority level i is the longest entry call by a lower level client task to a server whose priority ceiling is equal to or higher than the priority of i. The worst-case blocking time is the same whether the ceiling protocol is implemented directly or is approximated by giving the server an appropriately high priority. The essential difference between the direct implementation and the approximation method is that for the direct implementation, the server priority will be raised only when necessary. This tends to generate less blocking on average and hence better response time when aperiodic tasks have a priority below that of a server's ceiling; in addition, in transient overload situations, non-critical periodic tasks having a priority lower than a server's ceiling are less likely to miss their deadlines. In short, the direct implementation gives better average case behavior, especially when the entry calls are relatively long. When the entry calls are short compared with client task execution times, the performance difference is insignificant.

4. Conclusion

Ada tasking was intended to be used for real-time programming. However, the Ada tasking model represents a fundamental departure from the traditional cyclical executive model. Indeed, the dynamic preemption of tasks at runtime generates non-deterministic timelines that are at odds with the very idea of a fixed execution timeline required by a cyclical executive.

In this paper, we have reviewed some important results of priority scheduling theory. Together with Ada tasking, they allow programmers to reason with confidence about timing correctness at the tasking level of abstraction. As long as the analytic scheduling algorithms are supported by the runtime system and the resource utilization bounds on CPU, I/O drivers, and communication media are observed, the timing constraints will be guaranteed. Even if there is a transient overload, the tasks missing deadlines will be in a predefined order.

Although the treatment of priorities by the current Ada tasking model can and should be improved, most of the scheduling algorithms can be used today within the existing Ada rules if an appropriate coding and design approach is taken, and if schedulers are written to take full advantage of certain coding styles and the existing flexibility in the scheduling rules. Additional reports on how this can be done are in preparation at the Software Engineering Institute.

References

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

[2] Kleinrock, L.
Queueing Systems, Vol I.
John Wiley & Sons , 1975.

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

[4] Lehoczky, J. P., Sha L. and Strosnider, J.
Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment.
IEEE Real-Time System Symposium , 1987.

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

[6] Rajkumar, R., Sha, L., and Lehockzy J.P.
Real-Time Synchronization Protocols for Multiprocessors.
To appear in Proceedings of the IEEE Real-Time Systems Symposium , 1988.

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

[8] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
To appear in IEEE Transactions on Computers , 1987.

[9] Sprunt, B., Sha, L. and Lehoczky, J. P.
Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System.
Technical Report (in preparation) , 1988.

Table of Contents

List of Figures

List of Tables