Technical Report
CMU/SEI-89-TR-014
ESD-TR-89-022

Real-Time Scheduling Theory and Ada

Lui Sha
John B. Goodenough

April 1989

Technical Report
CMU/SEI-89-TR-014
ESD-TR-89-022
April 1989

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 support the management of concurrency in a priority-driven scheduling environment. In this paper, we 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. This paper is a revision of CMU/SEI-88-TR-33.[1] A shortened version is also being presented at the 1989 Ada-Europe Conference.

1. Introduction

1.1. Background

The Real-Time Scheduling in Ada Project at the Software Engineering Institute is a cooperative effort between the SEI, Carnegie Mellon University, system developers in industry, Ada vendors, and DoD agencies. It aims at applying the scheduling theory reviewed in this paper to the design and implementation of hard real-time systems in Ada. The scheduling algorithms and theories developed under this project and at Carnegie Mellon provide an analytical basis for understanding the timing behavior of real-time systems. The project is implementing these scheduling algorithms in an Ada runtime system, and is coding examples of real-time systems to evaluate the suitability of the whole approach using a generic avionics application, a generic missile application, and a generic inertial navigation system. This paper summarizes some of the scheduling approaches being studied and shows how they can be applied in an Ada context.

Traditionally, many real-time systems use cyclical executives to schedule concurrent threads of execution. Under this approach, a programmer lays out an 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 develop application code so that the compiled segments fit into the time slots of a cyclical executive and to ensure that the critical sections of different tasks do not interleave. Forcing programmers to schedule tasks by fitting code segments 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 nondeterministic

[1]The most important revisions affect our discussion of aperiodic tasks and our analysis of how to support the priority ceiling protocol.

timelines that are at odds with the very idea of a fixed execution timeline. This nondeterminism seems to make it impossible to decide whether real-time deadlines will be met. However, Ada's tasking concepts are well-suited to the analytic scheduling theories being considered in our project. In essence, these theories ensure that as long as the CPU utilization of all tasks lies below a certain bound and appropriate scheduling algorithms are used for the CPU and I/O processing, all tasks will meet their deadlines without knowing exactly when any given task will be running. Even if there is a transient overload, a fixed subset of *critical* tasks will still meet their deadlines as long as their CPU utilizations lie within the appropriate bound. The theories also deal with aperiodic processing requirements, mode changes, and jitter requirements. Applying these theories to Ada makes Ada tasking truly useful for real-time applications while also putting the development and maintenance of real-time systems on an analytic, engineering basis, making these 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 [Liu & Layland 73], 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.

Chapter 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 Chapter 2 by reviewing the problems of real-time synchronization. In Chapter 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 Chapter 4.

2. Scheduling Real-Time Tasks

In this chapter, 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 chapter 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 [Liu & Layland 73].

> **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
>
> $$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) = U(n)$$
>
> 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% [Lehoczky et al. 87]. 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 [Lehoczky et al. 87].

> **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.[2] 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_1 ($i = 1$) 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_2 ($i = 2$), 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

[2]It was shown in [Liu & Layland 73] 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.

time from 20 to 40, so the total processor utilization increases from 0.753 to 0.953. Since the utilization of the 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 $l = 3, k = 1$[3]
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[4] 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

[3]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. [Liu & Layland 73] showed that since the tasks meet their deadlines at least once within the period T_3, they will always meet their deadlines.

[4]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.

computation each. 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 indicating 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, a 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 for this example, all deadlines are met, but total machine utilization must be 95.3% or less instead of 100% or less. This doesn't mean the rate monotonic approach is less efficient. The capacity that isn't needed to service real-time tasks in the rate monotonic approach 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 task preemption needs to be taken into account. If S is the amount of time needed for a single scheduling action, then the total utilization devoted to scheduling is $2S/T_1 + 2S/T_2 + 2S/T_3$ since there are two scheduling actions per task. In the cyclic case, the, scheduling overhead is partly in the (very small) 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 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. During transient overload, we want to guarantee the deadlines of the tasks that are critical to the mission, even at the cost of missing the deadlines of non-critical tasks.

We consider a scheduling algorithm to be *stable* if there is a fixed set of tasks (the *stable* set) that always meet their deadlines even if the processor is overloaded. Tasks outside the stable set may miss their deadlines as the processor load increases or as task phasings change. To ensure that the critical tasks always meet their deadlines, it is sufficient for them to belong to the stable set of tasks.

The rate monotonic algorithm is a stable scheduling algorithm. Suppose there are *n* tasks. The stable set consists of the first *k* highest priority tasks that satisfy Theorem 1 or 2. For example, if the utilization of the first *k* tasks is less than 69.3%, the tasks will always meet their deadlines no matter what the total processor load is. Of course, which tasks are in the stable task set depends on the worst-case utilizations of the particular tasks considered. The important point is that the rate monotonic algorithm 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 [Sha et al. 86].

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%, 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:

Example 4: Let

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

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.

If periods are uniformly harmonic, i.e., if each period is an integral multiple of each shorter

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

period, the utilization bound of the rate monotonic algorithm is 100%.[6] 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 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 $(C + 2S)/T$. 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 [Lehoczky 87], 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 87] 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.

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

Compared with background service, the deferrable server algorithm typically improves aperiodic response time by a factor between 2 and 10 [Lehoczky 87]. 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 et al. 89]. 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.[7] 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.[8] 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.

Figure 2-1: Scheduling Both Aperiodic and Periodic Tasks

[7]Early refreshing is also possible under certain conditions. See [Sprunt et al. 89].

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

Simulation studies of the sporadic server algorithm [Sprunt et al. 89] show that in a lightly loaded system, aperiodic events are served 5-10 times faster than with background service, and 3-6 times faster than with polling. Figure 2-1, from [Sprunt et al. 89], 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) is given a period that allows it to execute as the highest priority task.[9] 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% 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%. 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. A server can resume its execution at its assigned priority, only when its budget is refreshed.

Figure 2-1 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%,[10] the deferrable and sporadic server response time is about 10% of the average background response time, while the average polling response time is about 65% 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%, the different server algorithms do not give significant performance improvement over background service.

[9]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.

[10]A 5% average aperiodic workload means that in the long run, the aperiodic requests consume about 5% of the CPU cycles, although the number of requests and their execution time vary from period to period and from request to request.

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 [Rajkumar et al. 88].

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 nonpreemptable 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, which means that at most one lower priority task can block a higher priority task [Goodenough & Sha 88, Sha et al. 87]. 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 [Sha et al. 87] 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 [Sha et al. 87]:

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

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 [Sha et al. 87].

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

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 inter-arrival 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.[11] 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.[12] 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: $C_3 = 100$; $T_3 = 350$; $B_3 = 0$; $U_3 = 0.286$

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

[12]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.

Using Theorem 4:

1. Task τ_1: Check $C_1 + B_1 \leq T_1$. Since $40 + 20 \leq 100$, task τ_1 is schedulable.

2. Task τ_2: Check whether either

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

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

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 [Kleinrock 75] 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. The results derived for this example show how the scheduling theory puts real-time programming on an analytic *engineering* basis.

3. Real-Time Scheduling in Ada

Although Ada was intended for use in building real-time systems, its suitability for real-time programming has been widely questioned. Many of these questions concern practical issues, such as the cost of performing a rendezvous, minimizing the duration of interrupt masking in the runtime system, providing efficient support for interrupt handling, etc. These problems are being addressed by compiler vendors who are aiming at the real-time market. More important are concerns about the suitability of Ada's conceptual model for dealing with real-time programming. For example, tasks in Ada run nondeterministically, 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 (due to priority inversion; see [Goodenough & Sha 88] for an example); and task priorities cannot be changed when application demands change at runtime. Fortunately, it appears that none of these problems presents insurmountable difficulties; solutions exist within the current language framework, although some language changes would be helpful to ensure uniform implementation support. The Real-Time Scheduling in Ada Project at the SEI is specifying coding guidelines and runtime system support needed to use analytic scheduling theory in Ada programs. 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.

The rest of this section summarizes the approach being taken by the project, and then shows how Ada's scheduling rules can be interpreted to support the requirements of rate monotonic scheduling algorithms.

3.1. Ada Real-Time Design Guidelines

The coding and design guidelines being developed by the SEI Real-Time Scheduling in Ada Project reflect a basic principle of real-time programming — write systems in a way that minimizes priority inversion; that is 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, called *clients*, that must exchange data among themselves. They do not call each other to exchange data. Instead, whenever they must read or write shared data or send a message, they call a *server* task. Each server task has a simple structure — an endless loop with a single select statement with no guards.[13] This structure models the notion of critical regions guarded by a semaphore; each entry is a critical region for the task calling the entry.

[13]The prohibition against guards simplifies the schedulability analysis and the runtime system implementation, but otherwise is not essential.

Client tasks are assigned priorities according to rate monotonic principles; that is, 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 (the next section explains why the runtime system is allowed to support the protocol), then give the server a low 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. Then, when a server is executing on behalf of a client task, no other client task will be allowed to call any server unless the client's priority is higher than the executing server's priority ceiling. If the client does not have a sufficiently high priority, its call will be blocked (i.e., the caller will be preempted just before the call would be placed on an entry's queue), and the server's priority will be raised to the priority of the blocked caller. This treatment of competing calls will ensure that a high priority task is blocked at most once by a server [Goodenough & Sha 88]. Moreover, because calls are preempted *before* they are queued, when the executing server completes its rendezvous, the highest priority blocked task will be able to execute. In effect, calls will be processed in priority order, and mutual deadlock caused by nested server calls will be impossible.

If the priority ceiling protocol is not directly supported by an Ada runtime system, the effect of the protocol can often nonetheless be achieved. Suppose a server is used to synchronize access to data shared by several tasks. The ceiling protocol requires that while a server is in rendezvous with a client, no other server be called by any client unless the client has a priority higher than the server's priority ceiling. This effect can be easily achieved using existing Ada runtime systems by assigning the server task a ceiling priority, i.e., a priority that is one greater than the priority of its highest priority caller. In this case, the server will be either waiting for a client or be serving a client at a priority that prevents other clients from executing and calling any server with an equal or lower priority ceiling. This approach avoids the prioritized queueing problem because there will never be more than one queued client task. It is also not hard to see why mutual deadlock will be impossible.

This simple approximation to the ceiling protocol will not work, however, if the server task suspends itself while in rendezvous. Such a suspension will allow client tasks with priorities lower than that of the suspended server to rendezvous with other servers; this violates the principle of the ceiling protocol. Such a suspension can be caused by either the server's need to synchronize with external I/O events in a uniprocessor or the server's need to rendezvous with another task in another processor in a multiprocessor. When a server task can suspend itself while in a rendezvous, care must be taken to ensure that no calls are accepted by other servers having the same or a lower ceiling priority. In addition, because queues can now develop, it is important that queued calls be serviced in priority order. Preventing inappropriate server calls and ensuring priority queueing can require quite complex code involving entry families, so this method of implementing the ceiling protocol is probably unsuitable when server tasks are allowed to suspend themselves.

From a schedulability viewpoint, the execution time spent in each rendezvous with a server that does not suspend itself is counted as part of the computing time, C_i, for the client task τ_i. Because the use of servers is a synchronization problem, Theorems 3 and 4 must be used to account for blocking time. Under the priority ceiling protocol, the maximum blocking time for a nonserver 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. (Note that this definition of blocking time means that even if a task, τ, makes no server calls, the schedulability analysis for τ must include blocking time for lower priority client tasks; see Example above.) If a server can suspend itself, it is sufficient to treat the suspension time as server execution time in the schedulability analysis; work is underway to see when less pessimistic treatment may be possible.

The worst-case blocking time for a client task is the same whether the ceiling protocol is supported directly by the runtime system or is approximated by giving the server an appropriately high priority. The essential difference between the direct implementation and the approximation method is that the server priority will be raised only when necessary when the direct implementation is used. 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, noncritical 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.

Despite the complications that arise when server tasks can suspend themselves, our point is that the schedulability theory can be readily applied to Ada programs, ensuring that deadlines are met even when timelines are nondeterministic. In the next section we discuss how to interpret Ada's scheduling rules so that Ada runtime systems can support rate monotonic scheduling algorithms directly.

3.2. On Ada Scheduling Rules

First of all, the Ada tasking model is well-suited, in principle, to the use of the analytic scheduling theories presented in this paper. When using these theories, a programmer doesn't need to know when tasks are running to be sure that deadlines will be met. That is, both Ada and the theory abstract away the details of an execution timeline. 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 server task that is already in rendezvous with a low priority task, the rendezvous can continue 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 program have no assigned priority, then the runtime system 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 rate monotonic 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 entry calls 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, or by suspending calling tasks just before they are queued. Of course, telling programmers to assign "scheduling priorities" to tasks but not to use pragma PRIORITY, surely says we are fighting the language rather than taking advantage of it. *But the important point is that no official revisions to the language are needed to take advantage of the scheduling theories and algorithms described in this paper and being developed by our project.* Here are the relevant Ada rules and appropriate ways of interpreting them:

- <u>CPU allocation: priorities must be observed</u>. 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 an entry 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 and avoids deadlock by blocking calls from medium priority tasks when certain low priority entry calls are already in progress. For example (see Figure 3-1), suppose a resource is guarded by a server task S, 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 because M is the highest priority task that is eligible to run and T is able to accept the call.

  <u>Solution:</u> There are several solutions to this problem. First of all, the priority ceiling protocol need not be applied to all called Ada tasks; it need only be applied to those tasks that are servers in the sense defined in the previous section. The simplest approach (in terms of living within the current interpretations of Ada rules) is to give each server task a unique priority that is higher than the priority of any task calling the server.

  Another approach is to implement the priority ceiling protocol directly for server tasks. This approach is clearly allowed by Ada's scheduling rules as long as the server task is given no assigned priority. A server task with no assigned priority can preempt any other task at any time, because Ada's scheduling rules, in essence, do not specify how tasks with no priority are scheduled. For example, task S is allowed to preempt task M just at the point where M is about to call T. Moreover, when a server with no assigned priority is in a rendezvous with a client task, the Ada rules say the server is executed with "at least" the priority of the client task. Because the priority ceiling rules require that the server task be executed with the priority of the calling task or of any blocked task (whichever is higher), the Ada rule allows the server to be executed with the (higher) priority of the blocked task. In the example, this means S, in effect, continues its rendezvous with M’s priority. When the rendezvous is complete, M’s call is allowed to succeed. Hence, the priority ceiling protocol can certainly be implemented directly when server tasks have no assigned priority.

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

A more aggressive interpretation of Ada rules is to note that a high priority task must preempt a lower priority task only when it is "sensible" to do so. Surely it is not "sensible" for a medium priority task to start its rendezvous with T if the effect could be to delay the execution of a high priority task unnecessarily.[14] In short, the priority ceiling protocol provides a set of rules saying when it is "sensible" to allow a higher priority task to run, and hence, these rules can be followed directly by the runtime system even when a server task has an assigned priority.

In short, there are several ways to argue that it is possible to support the priority ceiling protocol’s view of priorities within the current Ada rules.

- <u>Hardware task priority: always higher than software task priorities</u>. 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.

[14]Suppose S can call T (such a call models a nested critical region). If M’s rendezvous with T is allowed to start and H then calls S, H could be delayed until the completion of both S and T’s rendezvous; that is, H would be blocked by more than one lower priority task. This is one of the reasons for preventing M’s call.

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. In other words, use the scheduling theory to take into account the effect of this source of priority inversion.

- 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 when tasks of different priorities are waiting at open select alternatives.

    Solution: Because Ada allows, but does not require taking these priorities into account, ensure that the runtime system *does* use these priorities to decide which call to accept. Alternatively, if the priority ceiling protocol is used, there is never more than one waiting task to select.

  - Called task priority: only increased during rendezvous.

    Solution: Use the solutions discussed under "CPU allocation" above; that is, increase the priority of a server when the ceiling protocol says this is "sensible," or give the called task no priority at all and use priority ceiling rules to say when the server is allowed to execute, or give servers a priority higher than that of their callers.

- FIFO entry queues: Ada requires that the priority of calling tasks be ignored; calls must be serviced in their order of arrival, not in order of priority. Using FIFO queues rather than prioritized queues usually has a serious negative effect on real-time schedulability. FIFO queues must be avoided.

  Solution: As noted earlier, often it is possible to avoid FIFO queueing by preventing queues from being formed at all. If the runtime system does not prevent queues from forming, then entry families can, of course, be used to get the effect of prioritized queueing.

- 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 scheduling rules for a certain class of aperiodic servers demand that the priority of such a server be lowered when it is about to exceed the maximum execution time allowed for a certain interval of time, and be raised when its service capacity has been restored [Sprunt et al. 89].

  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 scheduling the task appropriately by, in effect, changing its priority.

From what our project has learned so far, it seems to be possible in practice to support analytic scheduling algorithms in Ada by using an enlightened interpretation of Ada's scheduling rules together with a combination of runtime system modifications and appropriate coding guidelines. 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 reword some of these rules so that priority-based scheduling is more clearly and uniformly supported.

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 nondeterministic 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 analytic scheduling algorithms are supported by the runtime system and 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, it seems that 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.

Acknowledgements

The authors thank Mark Klein and Mark Borger for pointing out why the priority ceiling protocol cannot be correctly simulated simply by giving a server task a high priority when the server can suspend itself during a rendezvous. In addition, we thank Anthony Gargaro for catching an error in an early draft of Example 3.

References

[Goodenough & Sha 88]
Goodenough, J. B., and Sha, L.
The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks.
Ada Letters, Special Issue: Proceedings of the Second International Workshop on Real-Time Ada Issues VIII(7), Fall, 1988.

[Kleinrock 75] Kleinrock, L.
Queueing Systems, Vol I.
John Wiley & Sons, 1975.

[Lehoczky 87] Lehoczky, J. P., Sha, L., and Strosnider, J.
Enhancing Aperiodic Responsiveness in a Hard Real-Time Environment.
In *Proceedings of the IEEE Real-Time Systems Symposium*. 1987.

[Lehoczky et al. 87]
Lehoczky, J. P., Sha, L., and Ding, Y.
The Rate Monotonic Scheduling Algorithm — Exact Characterization and Average Case Behavior.
Technical Report, Department of Statistics, Carnegie Mellon University, 1987.

[Liu & Layland 73] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM 20 (1):46-61, 1973.

[Rajkumar et al. 88]
Rajkumar, R., Sha, L., and Lehockzy J.P.
Real-Time Synchronization Protocols for Multiprocessors.
In *Proceedings of the IEEE Real-Time Systems Symposium*. 1988.

[Sha et al. 86] Sha, L., Lehoczky, J. P., and Rajkumar, R.
Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.
In *Proceedings of the IEEE Real-Time Systems Symposium*. 1986.

[Sha et al. 87] Sha, L., Rajkumar, R., and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
Technical Report, Department of Computer Science, Carnegie Mellon University, 1987.
To appear in *IEEE Transactions on Computers*.

[Sprunt et al. 89] Sprunt, B., Sha, L., and Lehoczky, J. P.
Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System.
Technical Report CMU/SEI-89-TR-11, Software Engineering Institute, Carnegie Mellon University (in preparation), 1989.

Table of Contents

List of Figures

List of Tables