(2)

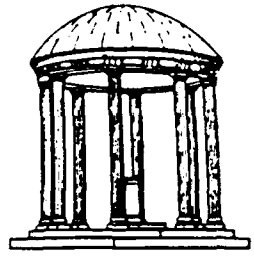
Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems

TR90-039

November, 1990

Kevin Jeffay

N00014-86-K-0680



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
jeffay@cs.unc.edu

November 1990

Abstract

We consider a characterization of a real-time system consisting as a set of sporadic tasks that share a set of serially reusable, single unit resources. Sporadic tasks are a generalization of periodic tasks and are well-suited for representing event driven processes. Resources are shared software objects, such as data structures. Tasks are composed of a sequence of phases. Each phase is a contiguous sequence of statements that require exclusive access to a resource. For an arbitrary instance of the model the goal is to determine if it is possible to schedule the tasks on a single processor such that:

- each invocation of each task completes execution at or before a well-defined deadline, and
- a resource is never accessed by more than one task simultaneously.

Our work makes two contributions to the theory of real-time scheduling and resource allocation. The first is the development of an on-line algorithm for sequencing a set of sporadic tasks on a uniprocessor such that the above criteria are met. The algorithm results from the integration of a synchronization scheme for access to shared resources with the *earliest deadline first* (*EDF*) algorithm of Liu and Layland. The result is deadline based scheduling algorithm in which phases of tasks that require exclusive access to resources have two types of deadlines: a *contending* deadline for the initial acquisition of the processor, and an *execution* deadline for subsequent execution. The algorithm is optimal with respect to the class of scheduling policies that do not use inserted idle time. The algorithm is optimal in the sense that it can schedule a set of tasks, without inserted idle time, whenever it will be possible to do so. The second contribution is a derivation of a set of relations on task parameters that are necessary and sufficient for a set of tasks to be schedulable. With these conditions one can efficiently decide whether it will be possible to schedule a set of tasks without executing or simulating the execution of the tasks. Our model for the analysis of processor scheduling policies is novel in that it incorporates minimum as well as maximum processing time requirements of tasks.

Index Terms: Analysis of algorithms, theory of deterministic processor and resource allocation, operating systems, real-time systems, scheduling theory.

Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
jeffay@cs.unc.edu

November 1990

1. Introduction

Real-time computer systems are loosely defined as the class of computer systems that must perform computations and I/O operations in a time frame defined by processes in the environment external to the computer. Real-time systems differ from more traditional multiprogrammed systems in that real-time systems have a dual notion of correctness. In addition to being logically correct, *i.e.*, producing the correct outputs, real-time systems must also be temporally correct, *i.e.*, produce the correct output at the correct time. In this paper we examine a processor and resource allocation problem for hard-real-time systems. Hard-real-time systems are real-time systems that require deterministic guarantees of temporal correctness. These are systems in which the cost of failing to be temporally correct is high. This high cost can be measured in monetary terms (*e.g.*, an inefficient use of raw materials in a process control system), aesthetic terms (*e.g.*, unrealistic output from a computer music or computer animation system), or possibly in human or environmental terms (*e.g.*, an accident due to untimely control in a nuclear power plant or fly-by-wire avionics system).

Hard-real-time systems are commonly structured as a set of tasks that are invoked repetitively. Two frequently studied classes of repetitive tasks are *periodic* tasks, *i.e.*, tasks that are invoked at constant intervals, and *sporadic* tasks, *i.e.*, tasks that are invoked at random but with a minimum inter-invocation interval [Mok 83]. In both cases, each invocation of a task must complete execution before a well-defined deadline. Our contribution to the study of repetitive, real-time workloads is the consideration of tasks that share a set of serially reusable resources. Our notion of a resource is a software object,

e.g., a data structure, that is shared among a group of tasks and must be accessed in a mutually exclusive manner. Operations on a shared resource therefore constitute a critical section. For example, within the context of a concurrent programming language in which shared data is encapsulated within a monitor [Hoare 74], a resource would be an individual monitor. We consider a characterization of a hard-real-time system as a set of sporadic tasks that share a set of serially reusable software resources. An invocation of a task will require exclusive access to a set of software resources. This paper examines the problem of scheduling sporadic tasks that share a set of software resources. The problem is to sequence a set of sporadic tasks on a uniprocessor such that in all cases — and in particular in the worst case — it is guaranteed that:

- each invocation of each task completes execution at or before its deadline, and
- a resource is never accessed by more than one task simultaneously.

Our work makes two contributions to the theory of real-time scheduling and resource allocation. The first is the development of an on-line algorithm for sequencing a set of *sporadic* tasks on a uniprocessor such that the above criteria are met. The algorithm results from the integration of a synchronization scheme for access to shared resources with the *earliest deadline first* (*EDF*) algorithm of Liu and Layland; a preemptive, priority-driven scheduling algorithm with dynamic priority assignment [Liu & Layland 73]. The algorithm is optimal with respect to the class of scheduling policies that do not use inserted idle time.[1] The algorithm is optimal in the sense that it can schedule a set of tasks, without inserted idle time, whenever it will be possible to do so. The second contribution is a derivation of a set of relations on task parameters that are necessary and sufficient for a set of tasks to be schedulable. With these conditions one can efficiently decide whether it will be possible to schedule a set of tasks without executing or simulating the execution of the tasks. Our model for the analysis of processor scheduling policies is novel in that it incorporates minimum as well as maximum processing time requirements of tasks. This work is part of a larger design system for hard-real-time systems [Jeffay 89a].[2]

---

[1] If tasks are scheduled by a discipline that allows itself to idle the processor when there exists a task with an outstanding request for execution, then that discipline is said to use *inserted idle time* [Conway et al. 67].

[2] For the remainder of this paper, we will use the terms *real-time* and *hard-real-time* interchangeably where it causes no confusion.

Several approaches to scheduling real-time tasks that share resources have been described in the literature [Leinbaugh 80, Mok 83, Mok et al. 87, Zhao et al. 87a,b, Jeffay 89b, Sha et al. 90, Chen & Lin 90]. Most consider the case where tasks are periodic and develop heuristic algorithms for scheduling the tasks. When tasks are periodic, Mok has shown that the problem of deciding whether or not it is possible to execute a set of tasks that use semaphores to enforce mutual exclusion is NP-hard [Mok 83]. In [Jeffay et al. 90] the more general problem of deciding whether or not it is possible to schedule a set of periodic tasks in a non-preemptive manner was also shown to be NP-hard in the strong sense. Moreover, it was shown that if an optimal non-preemptive scheduling algorithm exists for periodic tasks, then P = NP. If the times of all task invocations are known in advance, one can compute a schedule off-line and then apply the schedule at run-time [Xu & Parnas 90].

The following section presents our model of a real-time system in greater detail and defines the objective of our study. Section 3 examines the problem of scheduling tasks that use only a single resource. An optimal algorithm is developed for this special case. Section 4 generalizes this algorithm for tasks that share a set of resources. Section 5 discusses our results and revisits the assumptions and restrictions in our model.

2. System Model

We define a hard-real-time system as a set of sporadic tasks that share a set of serially reusable, single unit software resources. A sporadic task is a sequential program that is invoked in response to the occurrence of an event. An event is a stimulus that may be generated by processes external to the system (*e.g.*, an interrupt from a device) or by processes internal to the system (*e.g.*, the arrival of a message). We assume events are generated repeatedly with a (non-zero) lower bound on the duration between consecutive occurrences of the same event. Therefore, each sporadic task will be invoked repeatedly with a lower bound on the interval between consecutive invocations. Once invoked a task will execute to completion. Sporadic tasks are well-suited for implementing computational processes that are required to execute periodically (with a constant interval between activations) or in response to recurring asynchronous events. During the course of execution, a task may perform operations on shared data resources. Resources are serially reusable and must be accessed in a mutually exclusive manner. This model of software resources is motivated by the use of monitors for regulating access to shared data in process oriented concurrent programming languages such as Modula, Mesa, or Real-Time Euclid [Wirth 77, Lampson & Redell 80, Kligerman & Stoyenko 86].

Formally, we consider a real-time system that consists of a set of n sporadic tasks $\{T_1, T_2, ..., T_n\}$ and a set of m serially reusable, single unit resources $\{R_1, R_2, ..., R_m\}$. A task is described by a 3-tuple

$$T_i = (s_i, \{(c_{ij}, C_{ij}, r_{ij}) \mid 1 \leq j \leq n_i\}, p_i)$$

where:

s_i — the release time of task T_i: the time of the first invocation of task T_i,

$\{ (c_{ij}, C_{ij}, r_{ij}) \}$ — a set of n_i *phases* where for each phase:

- c_{ij} — the minimum computational cost: the minimum amount of processor time required to execute the j^{th} phase of task T_i to completion on a dedicated uniprocessor,
- C_{ij} — the maximum computational cost: the maximum amount of processor time required to execute the j^{th} phase of task T_i to completion on a dedicated uniprocessor,
- r_{ij} — the resource requirement: the resource (if any) that is required during the j^{th} phase of task T_i, and

p_i — the period of the task: the minimum time interval between invocations of task T_i.

Each task T_i is partitioned into a sequence of n_i disjoint phases. A phase is a contiguous sequence of statements that together require exclusive access to a resource. A task may have multiple phases that require the same resource. The resource required by task T_i during the j^{th} phase of its computation is represented by an integer r_{ij}, $0 \leq r_{ij} \leq m$. If $r_{ij} = k$, $k \neq 0$, then the j^{th} phase of T_i's computation requires exclusive access to resource R_k. For a given invocation of task T_i, in the interval between the time phase j commences execution and the time it completes execution, no other phase of a task that requires resource R_k may execute. If $r_{ij} = 0$, then the j^{th} phase of task T_i's computation requires no resources. In this case the j^{th} phase of task T_i imposes no mutual exclusion constraints on the execution of other tasks. Within the context of a concurrent programming language with monitors, if $r_{ij} \neq 0$, then the j^{th} phase of task T_i would consist of a call to an entry procedure of a monitor that encapsulates resource r_{ij}. If $r_{ij} = 0$, then the j^{th} phase of task T_i would consist of either code in the main body of the task or reentrant procedure code called by the main body of the task. Note that since different tasks may perform different operations on a resource (*e.g.*, call different monitor entry procedures), it is reasonable to assume that phases of tasks that access the same resource have varying computational

costs. A fundamental restriction is that each phase of each task will require access to at most one resource at a time. Other paradigms of resource usage and task decomposition will be discussed briefly in Section 5.

Throughout this paper we assume a discrete time model. In this domain all task parameters as well as all values of time are expressed as integer multiples of some indivisible time unit. Without loss of generality, assume these quantities are integers. Moreover, we assume throughout that tasks are sorted in non-decreasing order by period. For any pair of tasks T_i and T_j, if $i > j$, then $p_i \geq p_j$. The *index* of a task refers to its position in this sorted list.

The behavior of a sporadic task T_i is given by the following rules. Let t_k be the time of the k^{th} invocation of task T_i.

i) The initial invocation of task T_i occurs at time $t_1 = s_i$.

ii) If task T_i has period p_i, then for all $k \geq 1$, the $(k+1)^{st}$ invocation of T_i occurs at time $t_{k+1} \geq t_k + p_i \geq s_i + kp_i$.

iii) Each invocation of task T_i consists of the execution of n_i phases in sequence. The execution of an invocation of T_i commences in phase 1. The j^{th} phase of each execution of T_i does not commence until the $(j-1)^{st}$ phase has terminated.

iv) Execution of the j^{th} phase of task T_i requires at least c_{ij} units of processor time and at most C_{ij} units of processor time, $C_{ij} \geq c_{ij} \geq 0$.

v) The k^{th} invocation of T_i must be completed no later than time $t_k + p_i$. This time is commonly referred to as the *deadline* of the k^{th} invocation of task T_i.

If a phase of a task requires a resource then the computational cost of the phase represents only the cost of using the required resource and not the cost (if any) of acquiring or releasing the resource. A minimum cost of zero indicates that a phase of a task is possibly optional. (For example, the execution of a phase of a task may be dependent on the outcome of the evaluation of a boolean expression.)

The "period" of a sporadic task is simply the minimum time between any two successive invocations of the task. In general an arbitrary amount of time may lapse between successive invocations of a task. A sporadic task is a generalization of the more commonly studied periodic task [Liu & Layland 73]. We assume sporadic tasks are independent in the sense that the time of a task's invocation is dependent only upon the time of its last

invocation and not upon those of any other task. Once released, a sporadic task will be invoked an unbounded number of times.

If the k^{th} invocation of task T_i occurs at time t, then the closed interval $[t, t+p_i]$ is called the k^{th} *invocation interval*, or simply an *invocation interval*, of task T_i. If task T_i is invoked at time t and does not complete execution at or before time $t + p_i$, then we say that T_i has *failed*. A set of sporadic tasks τ is said to be *feasible* on a uniprocessor if it is possible to schedule τ on a uniprocessor such that:

- no task fails, *i.e.*, every invocation of every task completes execution at or before the end of its invocation interval, and
- for each task T_i, and for all phases j, $1 \leq j \leq n_i$, if $r_{ij} \neq 0$, then the j^{th} phase of each invocation of T_i has exclusive access to the resource $R_{r_{ij}}$ from the time the phase commences execution until the phase terminates execution.

A scheduling algorithm *succeeds* in scheduling a set of tasks if it can sequence the tasks such that both criteria above will be met. A scheduling algorithm is said to be *optimal* for a uniprocessor if it can succeed for any task set that is feasible on a uniprocessor. Our goal is to develop an algorithm that can sequence all feasible sets of tasks on a uniprocessor.

The characteristics of our real-time workload model motivate the consideration of *on-line* scheduling algorithms for sequencing the tasks. This is because it will not be possible to generate a schedule off-line if all invocation times of tasks are unknown. Given the possibly non-deterministic manner in which a sporadic task may be invoked, it is possible for this to be the case. In developing a scheduling algorithm, we assume that in principle tasks are preemptable at arbitrary points. However, the requirement of exclusive access to resources places two restrictions on the preemption and execution of tasks. For all tasks i and k, if $r_{ij} = r_{kl}$ and $r_{ij}, r_{kl} \neq 0$, then (1) the j^{th} phase of task T_i may neither preempt the l^{th} phase of task T_k, nor (2) execute while the l^{th} phase of task T_k is preempted.

Lastly, in the following sections it will be useful to distinguish between tasks, and phases of tasks, that share resources with other tasks and those that do not. If a task (phase) never requires a resource then that task (phase) is called a *non-resource requesting task (phase)*. If a task (phase) ever requires a resource it is called a *resource requesting task (phase)*.

3. Single Phase Task Systems

We first consider the problem of scheduling sporadic tasks that consist of only a single phase. As will be shown in Section 4, the general problem of scheduling tasks with multiple phases can largely be reduced to the problem of scheduling tasks with only a single phase.

The following sub-section establishes conditions that are necessary for a set of single phase sporadic tasks to be feasible in the absence of inserted idle time. (In Section 5 we will briefly comment on the problem of scheduling sporadic tasks with inserted idle time.) Section 3.2 then develops an algorithm for scheduling such tasks and demonstrates its optimality.

3.1 Feasibility Conditions for Single Phase Task Systems

Consider a set of single phase sporadic tasks $\{T_1, T_2, ..., T_n\}$, where $T_i = (s_i, (c_i, C_i, r_i), p_i)$,[3] that share a set of m serially reusable, single unit resources $\{R_1, R_2, ..., R_m\}$. It will be useful to refer to the period of the "shortest" task that uses resource R_i. For resource R_i, let P_i represent this period. That is,

$$P_i = \underset{1 \le j \le n}{\text{MIN}} (p_j \mid r_j = i).$$

We first demonstrate that the feasibility of a set of sporadic tasks is not a function of their release times. The following Lemma demonstrates that if a set of tasks is feasible, then the tasks will be feasible for any combination of release times.

Lemma 3.1: Let τ be a set of sporadic tasks. If τ is feasible then the set of sporadic tasks τ' obtained from τ by replacing the release times of tasks with arbitrary values will also be feasible.

Proof: By the definition of a sporadic task, an arbitrary amount of time may elapse between the end of one invocation and the start of the next. Therefore, after all tasks in τ have been released, there can exist a time t such that a task, or group of tasks, in τ are invoked at time t, and such that all task invocations that occur prior to time t with deadlines

---

[3] Since tasks consist of only a single phase, the second subscript on the parameters C, c, and r will be omitted.

after t, have completed execution at or before time t. That is, if the task invocation(s) occurring at time t did not exist then the processor would have been idle for some non-zero length interval starting at t. At time t, τ is effectively "starting over" with a set of "release times" that are unrelated to the initial release times. Therefore, if τ is feasible then any set of tasks derived from τ by replacing the release times with arbitrary values must also be feasible. □

The following theorem establishes necessary conditions for a set of single phase tasks to be feasible.

Theorem 3.2: Let τ be a set of single phase sporadic tasks $\{T_1, T_2, ..., T_n\}$, sorted in non-decreasing order by period, that share a set of m serially reusable, single unit resources $R_1, R_2, ..., R_m$. If τ can be scheduled on a uniprocessor without inserted idle time, then:

1) $\sum_{i=1}^{n} \frac{C_i}{p_i} \leq 1,$

2) $\forall i,\ 1 < i \leq n$ and $r_i \neq 0$; $\forall L,\ P_{r_i} < L < p_i$:

$$L \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor C_j .$$

Informally, condition (1) can be thought of as a requirement that the processor not be overloaded. If a task T has maximum cost C and period p, then C/p is the least upper bound on the fraction of processor time consumed by T over the lifetime of the system (*i.e.*, the worst case utilization of the processor by T). The first condition simply stipulates that the cumulative processor utilization cannot exceed unity. The right hand side of the inequality in condition (2) is a least upper bound on the processor demand that can be realized in an interval of length L starting at the time an invocation of a resource requesting task T_i is scheduled, and ending sometime before the end of the invocation interval. This interval is illustrated in Figure 3.1. Figure 3.1 shows an invocation interval of task T_i. Task T_i is invoked at time t and is scheduled at time t'. The striped rectangle in the invocation interval represents the execution of task T_i. This invocation must complete execution at or before time $t + p_i$.

For a set of tasks to be feasible, the processor demand in this interval must always be less than or equal to the length of the interval. If this is not the case then a task can fail. Although condition (2) is semantically similar to the requirement that the processor not be

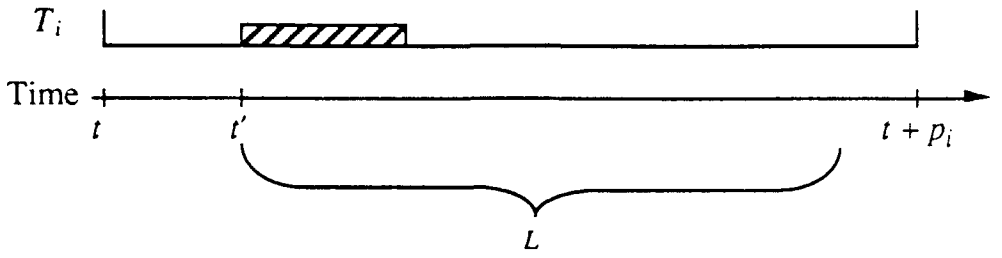


Figure 3.1

over-utilized, we will later demonstrate that conditions (1) and (2) are in fact not related. The intuition behind conditions (1) and (2) will be developed further in the proof of Theorem 3.2.

Proof: By Lemma 3.1, to show that conditions (1) and (2) are necessary for feasibility, it suffices to demonstrate that there exist release times for which these conditions are necessary for τ to be feasible. We first show that condition (1) is necessary.

For a set of tasks τ, the *achievable processor demand* in the time interval $[a, b]$, written $d^*_{a,b}$, is defined as the maximum amount of processing time required by τ in the interval $[a, b]$ to complete all invocations of tasks with deadlines in the interval $[a, b]$. That is, $d^*_{a,b}$ is the processing time required, in the worst case, by τ in the interval $[a, b]$ to ensure that no task fails in the interval $[a, b]$. The worst case occurs when tasks are periodic from point a onward. If a set of tasks τ is feasible, then for all a and b, $a < b$, it follows that $d^*_{a,b} \leq b - a$.

For all i, $1 \leq i \leq n$, let $s_i = 0$ and let $t = p_1 p_2 \ldots p_n$. In the interval $[0, t]$, $\frac{t}{p_i} C_i$ is the maximum amount of processor time that must be allocated to task T_i to ensure that T_i does not fail in the interval $[0, t]$. In the interval $[0, t]$ the achievable processor demand, $d^*_{0,t}$, is therefore

$$d^*_{0,t} = \sum_{i=1}^{n} \frac{t}{p_i} C_i .$$

If τ is feasible then it must be the case that $d^*_{0,t} \leq t$, hence

$$\sum_{i=1}^{n} \frac{t}{p_i} C_i \leq t$$

$$\sum_{i=1}^{n} \frac{C_i}{p_i} \leq 1.$$

For condition (2) choose a task T_i, $1 < i \leq n$, such that $r_i \neq 0$ and $p_i > P_{r_i}$. Let $s_i = 0$ and $s_j = 1$ for all j, $1 \leq j \leq n$, $j \neq i$. This gives rise to the pattern of initial task invocations shown in Figure 3.2. Initially only task T_i is eligible for execution. Since inserted idle time is not allowed, task T_i must execute in the interval [0,1]. For all L, $L > P_{r_i}$, the interval $[0, L]$ contains at least one invocation of some task T_k with $r_k = r_i$. Since task T_k shares a resource with task T_i and since this resource is in use by task T_i at time 1, the initial invocation of task T_k may not be scheduled until after the invocation of T_i made at time 0 has completed execution. Therefore, to ensure that the initial invocation of task T_k does not fail, the initial invocation of task T_i must be completed before time $p_k + 1 = P_{r_i} + 1$. Hence for this choice of release times, for all L, $P_{r_i} < L < p_i$, in the interval $[0, L]$ the achievable processor demand, $d^*_{0,L}$, is

$$d^*_{0,L} = C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor C_j .$$

The demand consists of the maximum cost of executing the initial invocation of task T_i plus the achievable processor demand due to tasks 1 - i–1 in the interval $[1, L]$. (Note that tasks with periods greater than or equal to p_i have no invocation intervals contained in the interval $[1, L]$ and hence can not fail in the interval $[1, L]$. Therefore these tasks do not contribute to the achievable processor demand in the interval $[1, L]$.) For τ to be feasible it must be

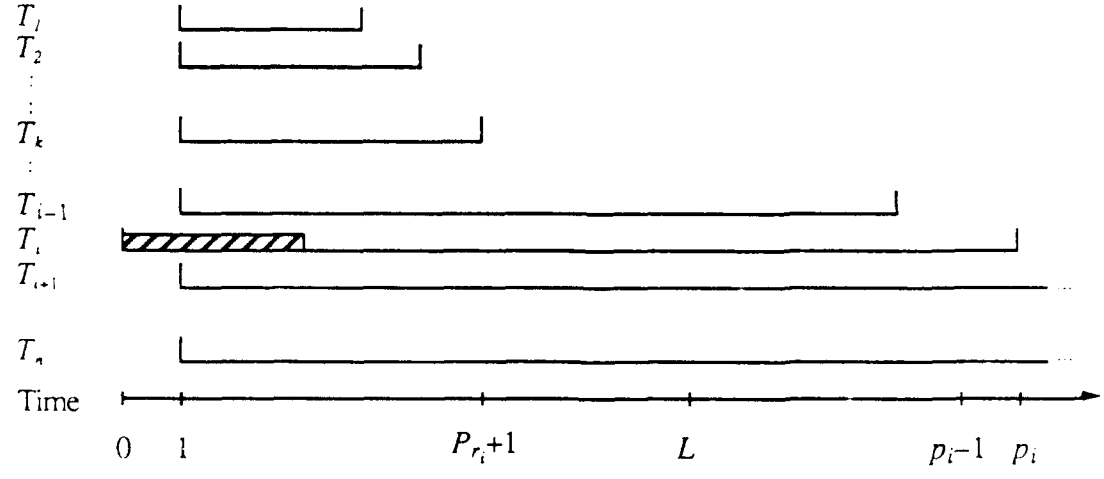


Figure 3.2

the case that $L \geq d^*_{0,L}$, hence

$$L \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor C_j . \qquad \square$$

Although seemingly arbitrary, the constructions used in the proof of Theorem 3.2 precisely characterize the worst case interleavings of task invocations for a set of sporadic tasks. In essence, it will be shown in Section 3.2 that if a set of tasks can be scheduled when interleaved as shown above, then the tasks are indeed feasible. The notion of a worst case interleaving is important as Lemma 3.1 indicates that such an interleaving can always occur during the execution of any task set.

Note that a set of single phase sporadic tasks τ in which $r_i = 0$, for $1 \leq i \leq n$, corresponds to a set of tasks with no resources and hence no mutual exclusion constraints. In such a system a task would, in principle, be preemptable at any time during its execution by any other task. If $r_i = 0$, for $1 \leq i \leq n$, then condition (2) is void (the quantification of i is empty) and only condition (1) is necessary for feasibility. This agrees with the results reported in [Jeffay 89a, Liu & Layland 73] for the preemptive scheduling of periodic and sporadic tasks. Similarly, if tasks require resources but the resources are not shared (*i.e.*, there is only a single task that requests each resource) then condition (2) is again void (the quantification of L in condition (2) is empty for all tasks i). At the other extreme, a set of single phase sporadic tasks in which for all i, $1 \leq i \leq n$, $r_i = k$, for some $k \neq 0$, corresponds to a set of tasks that all share a single resource. Such single phase tasks must be scheduled non-preemptively. In this case condition (2) applies to all tasks and the feasibility conditions agree with those reported in [Jeffay et al. 90] for the non-preemptive scheduling of sporadic tasks.

3.2 Scheduling Single Phase Task Systems

Our goal is to develop an algorithm that will sequence a set of single phase sporadic tasks on a single processor whenever it is possible to do so. Such an algorithm must ensure that (1) all task invocations complete execution before the end of their respective invocation intervals and that (2) the mutual exclusion constraints on the execution of resource requesting tasks are respected. It is the latter requirement that motivates the development of a new scheduling policy. Our approach is to incorporate a synchronization protocol for mutual exclusion into an existing real-time scheduling policy.

The basis of the new scheduling policy is the preemptive *earliest deadline first* (EDF) algorithm [Liu & Layland 73]. The EDF scheduling algorithm works as follows. When a task is invoked, if the resource the task requires is in use by another task, then the requesting task is said to be *blocked*; otherwise the task is said to be *ready*. When an invocation of a task is executing on a processor, the task is *executing*. If a task is preempted while executing then it returns to the ready state. After completion of an invocation, and prior to the first invocation, a task is *terminated*. If task T_i is invoked at time t, then a scheduler must ensure that T_i completes execution at or before its deadline at time $t + p_i$. The EDF scheduling discipline dictates that at all points in time, the ready task with the nearest deadline should be executing. An EDF scheduler makes scheduling decisions (dispatches tasks) whenever a task is invoked or completes an invocation. At each of these scheduling points, an EDF scheduler dispatches the ready task with the nearest deadline; preempting the previously executing task if necessary. Ties between tasks with identical deadlines are broken arbitrarily. Both the task selection process and the process of dispatching a task are assumed to take no time in our discrete time system. Our consideration of an EDF policy is motivated by the fact that it has been shown to be an optimal policy both when tasks have no preemption or execution constraints [Liu & Layland 73] and when preemption is completely disallowed [Jeffay et al. 90]. The problem currently under consideration lies between these two extremes.

The EDF scheduling discipline can be extended to ensure exclusive access to shared resources by re-examining the concept of an execution deadline. If tasks share resources then when a resource requesting task T_i is invoked, it is no longer sufficient for the invocation to complete execution within p_i time units. It can be the case that a resource requesting task must complete execution *before* the end of its current invocation interval. This situation can occur when an invocation of a task with a deadline becomes blocked. For example, consider the problem of scheduling the following task set according to a naive application of the traditional preemptive EDF discipline: (recall T = (*release time*, (*minimum cost*, *maximum cost*, *resource*), *period*))

$$\tau = \{ \; T_1 = (1, (1, 1, 1), 4) \\ T_2 = (2, (1, 2, 0), 10) \\ T_3 = (0, (1, 3, 1), 20) \; \}.$$

τ consists of three single phase tasks and 1 shared resource (R_1). The initial interleaving of invocations is illustrated in Figure 3.3.

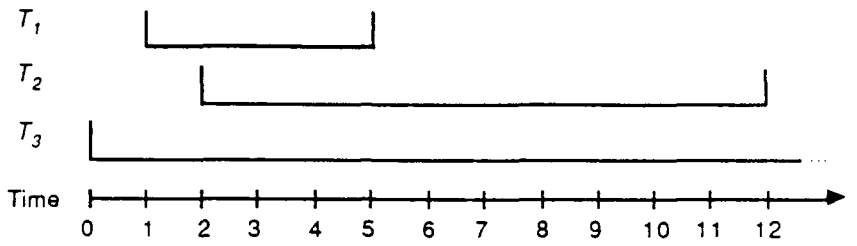


Figure 3.3

The initial invocation of task T_3 occurs at time 0. Since inserted idle time is not allowed, task T_3 will be scheduled at time 0 as shown in Figure 3.4. (In Figure 3.4 striped rectangles denote execution with resource R_1. Unfilled rectangles represent execution with resource R_0 (*i.e.*, execution with no resource). An execution rectangle open on the right side indicates that the execution was preempted. An execution rectangle open on the left side indicates that a previously preempted execution is resumed.) At time 1 task T_1 has the nearest deadline. However, since T_1 requires the resource that, in the worst case, is in use by task T_3 at time 1, task T_1 is blocked by task T_3. Therefore, task T_3 continues execution at time 1. At time 2, task T_2 has a nearer deadline than the executing task T_3. Since $r_2 \neq r_3$, one might be tempted to allow task T_2 to preempt task T_3. However, as illustrated in Figure 3.4, such a decision can cause task T_1 to fail at time 5. At time 1, it is no longer sufficient for the invocation of task T_3 occurring at time 0 to be completed by its nominal deadline at time 20. Since tasks T_1 and T_3 share a resource, when task T_1 is invoked at time 1, the invocation of task T_3 occurring at time 0 must now be completed no later than time 5: the initial deadline of task T_1. (Of course the initial invocation of task T_3 must actually be completed by time $5 - C_1 = 4$. It will turn out, however, that this is not a useful observation.)

The challenge is to quantify precisely *when* a task invocation must be completed. We claim that an invocation of a resource requesting task should have *two* notions of a deadline: one for the initial acquisition of the processor, and one for subsequent execution. Specifically,

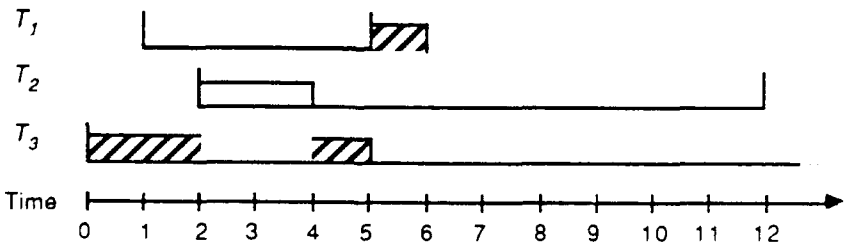


Figure 3.4

when a resource requesting task T_i is invoked at time t_r, the invocation should have an initial deadline equal to $t_r + p_i$ as in traditional EDF scheduling. This deadline will be referred to as the *initial* or *contending* deadline. Let t_s be the time that the invocation of task T_i occurring at time t_r is first scheduled (commences execution). After time t_s, the invocation of task T_i should have a deadline at time MIN($t_r + p_i$, $(t_s + 1) + P_{r_i}$). Thus, when a scheduler first dispatches an invocation of task T_i, the scheduler will potentially assign T_i a nearer deadline. This deadline will be referred to as the *execution* deadline. Since we assume a discrete time domain, a resource requesting task T_i has a contending deadline at all points in time in the closed interval $[t_r, t_s]$ and, assuming $C_i > 1$, has an execution deadline at all points in the closed interval $[t_s+1, t_c-1]$, where t_c is the time that the execution of the invocation terminates. (In the interval between the completion of one invocation and the start of the next, a task logically has a deadline of infinity.) This is illustrated in Figure 3.5 which plots the deadline of an invocation of a task T_i that has an execution deadline of $(t_s + 1) + P_{r_i}$ as a function of time. If a resource consuming task has a maximum computational cost of 1, then it will never have an execution deadline. Non-resource requesting tasks require no special treatment. If a non-resource requesting task T_j is invoked at time t_r, the invocation will have a deadline at time $t_r + p_j$ for the duration of its execution. We will refer to our scheme of dynamically altering the deadlines of resource requesting tasks as the *dynamic deadline modification* (DDM) strategy.

The application of the dynamic deadline modification strategy to the tasks in the previous example results in the non-preemptive schedule illustrated in Figure 3.6. Under this policy the initial invocation of task T_3 has a contending deadline at time 20 as before. However, once task T_3 is scheduled it will execute with a deadline equal to MIN($0 + p_i$, $(0 + 1) + P_{r_i}$)

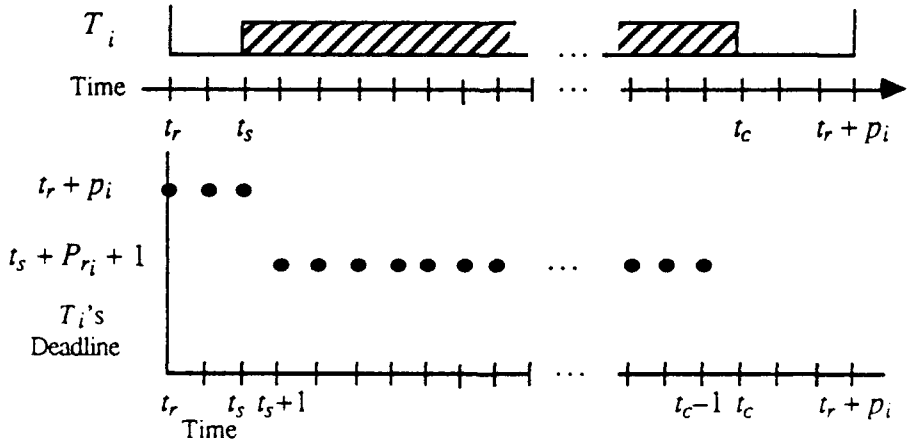


Figure 3.5

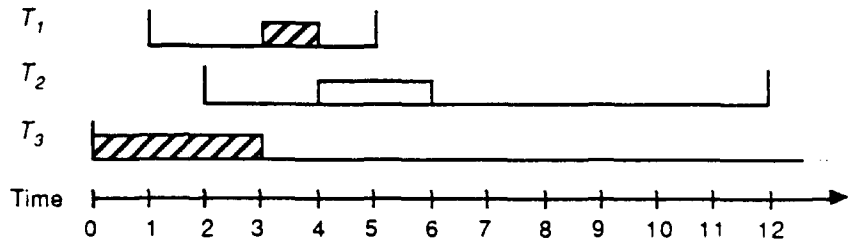


Figure 3.6

= MIN(20, 1 + 4) = 5. That is, at times 1 and 2, task T_3 has a deadline at time 5. When task T_2 is invoked at time 2, its invocation will have an initial deadline at time $2 + p_2 = 12$. At time 2, task T_3 now has a nearer deadline than task T_2 and hence an EDF scheduler will not allow T_2 to preempt T_3 at time 2.

The imposition of separate deadlines for execution and initial acquisition of the processor ensures that blocked tasks become unblocked (ready) as soon as possible. Although an invocation of a resource requesting task may now execute with a deadline that occurs before the end of the invocation interval, this "deadline" is indeed a deadline. We will eventually show that the task system can fail if an invocation of a resource requesting task does not complete execution by its execution deadline. That is, there can exist an invocation of a task that is not completed at or before the end of its invocation interval. Note that the deadline modification rule in the proposed algorithm is pessimistic in the sense that it requires all invocations of resource requesting tasks to execute with a modified deadline as soon as any blocking can possibly occur (*i.e.*, immediately after they are scheduled). In particular, resource requesting tasks execute with a modified deadline independently of whether or not any blocking can actually occur. A more optimistic approach, for example, would be to modify the deadline of a resource requesting task only when the execution of the task actually blocks some other task. In Section 5 we show that such an optimistic scheduling strategy is inferior to the pessimistic strategy we are proposing.

A final point to address concerns the mutual exclusion constraints on access to resources. As we will soon demonstrate, the combination of EDF scheduling with the dynamic deadline modification strategy is sufficient for ensuring tasks access resources in a mutually exclusive manner. There is, however, one subtlety in the case that there exist multiple outstanding invocations with the earliest deadline. To guarantee that the mutual exclusion constraints are respected, when there exist multiple tasks with outstanding invocations with the earliest deadline, a scheduler must:

- allow the currently executing task to continue execution if it has the earliest deadline,
- select a task with an outstanding invocation that has been preempted before selecting any task whose outstanding invocation has not commenced execution.[4]

We will refer to the combination of an EDF task selection policy with our dynamic deadline modification strategy and tie breaking rules as *earliest deadline first* scheduling with *dynamic deadline modification* (EDF/DDM). We validate the design of the EDF/DDM scheduling policy by demonstrating that it is an optimal discipline (with respect to the class of disciplines that do not use inserted idle time) for scheduling a set of single phase tasks that share a set of resources. To prove optimality it suffices to show that the satisfaction of conditions (1) and (2) from Theorem 3.2 is sufficient for ensuring that the EDF/DDM discipline will succeed in scheduling a set of tasks with shared resources. To demonstrate that the discipline succeeds in scheduling a set of tasks it must be shown that (1) all invocations of all tasks complete execution before the end of their respective invocation intervals and that (2) the mutual exclusion constraints on the execution of resource requesting tasks are respected. The following lemma demonstrates that the EDF/DDM scheduling discipline enforces the mutual exclusion constraints on the execution of resource requesting tasks.

Lemma 3.3: The EDF/DDM scheduling discipline ensures that resources are accessed in a mutually exclusive manner.

Proof: It suffices to show that a task that requires resource R_j can neither preempt another task that requires resource R_j nor execute while such a task is preempted when scheduled by the EDF/DDM scheduling discipline.

Let task T_i be a resource R_j requesting task. Let t_s be a point in time at which an invocation of task T_i commences execution. Let $t > t_s$ be a point in time at which this invocation is either executing or is preempted. Let T_k be a resource R_j requesting task with an invocation that is contending for the processor at time t. Let t_r be the time at which this invocation by task T_k was made. Note that under the EDF/DDM scheduling discipline, in order for task T_k to preempt task T_i or to execute while T_i is preempted, it must be the case that $t_s < t_r \leq t$ (and that $t_r + p_k < t_s + p_i$) as shown in Figure 3.7.

---

[4] Note that the first tie breaking rule ensures that at any point in time there can exist only one preempted task with the earliest deadline.

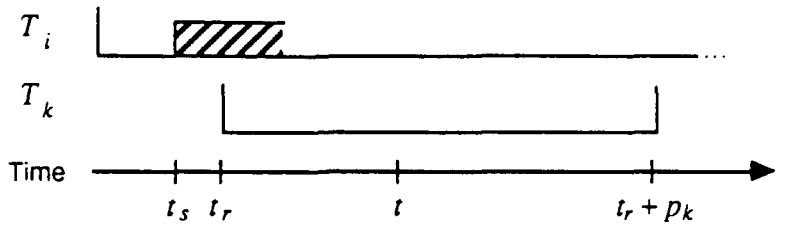


Figure 3.7

The invocation of task T_k occurring at time t_r will have an initial deadline at time $d_k = t_r + p_k$. Since task T_i is scheduled at time t_s, its invocation must have a deadline no later than at time $d_i = t_s + P_j + 1 \leq t_s + p_k + 1$. Since $t_s < t$, it follows that $d_i \leq d_k$. If $d_i < d_k$, then the invocation of task T_k occurring at time t_r will not be scheduled until after the invocation of task T_i occurring at time t_s has completed execution. If $d_i = d_k$, then since the EDF/DDM scheduling discipline gives priority to the currently executing task and then to preempted tasks, task T_k will again not be scheduled until after the outstanding invocation of task T_i has completed execution. Therefore, a task that requires resource R_j can neither preempt another resource R_j requesting task nor execute while such a task is preempted. □

Theorem 3.4: Let τ be a set of single phase sporadic tasks $\{T_1, T_2, ..., T_n\}$, sorted in non-decreasing order by period, that share a set of m serially reusable, single unit resources $R_1, R_2, ..., R_m$. The EDF/DDM discipline will succeed in scheduling τ if conditions (1) and (2) from Theorem 3.2 hold.

Proof: Lemma 3.3 has shown that independently of the conditions necessary for feasibility, the EDF/DDM scheduling discipline maintains the mutual exclusion constraints on the execution of resource requesting tasks. It remains to show that the use of the EDF/DDM scheduling discipline guarantees that tasks will not fail if conditions (1) and (2) of Theorem 3.2 hold. This will be shown by contradiction.

Assume the contrary, *i.e.*, that conditions (1) and (2) of Theorem 3.2 hold and yet a task fails at some point in time when τ is scheduled by the EDF/DDM algorithm.

For a set of tasks τ, the *actual processor demand*, or simply the *processor demand*, in the interval $[a, b]$, written $d_{a,b}$, is defined as the least upper bound on the amount of processing time actually required by τ in the time interval $[a, b]$ to ensure that no task fails in the interval $[a, b]$. If a set of tasks τ is feasible, then for all a and b, $a < b$, it follows that $d_{a,b} \leq d^*_{a,b} \leq b - a$. The proof proceeds by deriving upper bounds on the actual processor

demand (*i.e.*, the achievable processor demand) for an interval ending at the time at which a task fails.

Let t_d be the earliest point in time at which a task fails. τ can be partitioned into three disjoint subsets A_1, A_2, and A_3, where

A_1 = the set of tasks that have an invocation with an initial deadline at time t_d,

A_2 = the set of tasks that have an invocation occurring prior to time t_d with initial deadline after t_d, and

A_3 = the set of tasks not in A_1 or A_2.

Tasks in A_3 either have a release time greater than t_d, or are not invoked immediately prior to time t_d. As will soon become apparent, to bound the actual processor demand prior to t_d, it suffices to concentrate on the tasks in A_2. Let $b_1, b_2, ..., b_k$ be the invocation times immediately prior to t_d of the tasks in A_2. There are two main cases to consider.

Case 1: None of the invocations of tasks in A_2 occurring at times $b_1, b_2, ..., b_k$ are scheduled prior to time t_d.

Let t_0 be the end of the last period in which the processor was idle. If the processor has never been idle let $t_0 = 0$. In the interval $[t_0, t_d]$, the actual processor demand is the total processing requirement of tasks that are invoked at or after time t_0, with deadlines at or before time t_d. This gives

$$d_{t_0,t_d} \leq d^*_{t_0,t_d} = \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor C_j .$$

Since there is no idle period in the interval $[t_0, t_d]$ and since a task fails at t_d, it must be the case that $d_{t_0,t_d} > t_d - t_0$. Therefore

$$t_d - t_0 < \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor C_j \leq \sum_{j=1}^{n} \frac{t_d - t_0}{p_j} C_j ,$$

and hence

$$1 < \sum_{j=1}^{n} \frac{C_j}{p_j} .$$

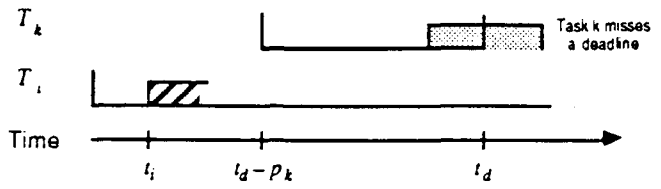


Figure 3.8

However, this is a contradiction of condition (1). Therefore, if conditions (1) and (2) hold and the EDF/DDM scheduling discipline fails to schedule τ, then an invocation of at least one task in A_2 must have been scheduled prior to t_d.

Case 2: Some of the invocations of tasks in A_2 occurring at times $b_1, b_2, \ldots, b_k$ are scheduled prior to time t_d.

Let T_i be the last task in A_2 to execute prior to t_d. Let t_i be the point in time at which the invocation of T_i containing the point t_d commences execution (is scheduled for the first time). Note that because of deadline-based scheduling, if a task T_k fails at time t_d then it must be the case that $t_i < t_d - p_k$. That is, the invocation that fails at time t_d is contained within the interval $[t_i, t_d]$ as shown in Figure 3.8.

We will show that if the invocation interval of task T_i containing the point t_d is scheduled prior to time t_d, then there must have existed enough processor time in the interval $[t_i, t_d]$ to schedule all invocations of tasks occurring after time t_i with deadlines at or before time t_d.

There are two sub-cases to consider depending on whether or not the invocation of task T_i scheduled at time t_i has an execution deadline less than or equal to time t_d. If this is the case then the invocation of task T_i scheduled at time t_i must be completed at or before time t_d.

Case 2a: The invocation of task T_i scheduled at time t_i has an execution deadline less than or equal to time t_d.

For this case to hold, since T_i is in A_2, task T_i must be a resource requesting task. We proceed by deriving the achievable processor demand for the interval $[t_i, t_d]$. If a task fails at time t_d then the following facts hold for Case 2a:

i) Other than task T_i, no task with period greater than or equal to $t_d - t_i$ executes in the interval $[t_i, t_d]$.

Since an invocation of task T_i is scheduled at time t_i and has an execution deadline less than or equal to t_d, every other task scheduled in $[t_i, t_d]$ must have had an initial deadline at or before t_d. Therefore, if an invocation of a task T_j,

with period greater than or equal to $t_d - t_i$, executes in the interval $[t_i, t_d]$, then this invocation of T_j must have been available for execution at time t_i. Consequently, since the invocation in question of task T_i had an initial deadline greater than time t_d, the EDF/DDM algorithm would have chosen task T_j before T_i in the interval $[t_i, t_d]$. Therefore, no task with period greater than or equal to $t_d - t_i$ executes in the interval $[t_i, t_d]$.

ii) Other than task T_i, no task which executes in $[t_i, t_d]$ could have been invoked at time t_i.

Again, other than T_i, every task that executes in $[t_i, t_d]$ has an initial deadline at or before t_d. Therefore, if a task $T_{i'}$ that executes in $[t_i, t_d]$ had been invoked at t_i, the EDF/DDM algorithm would have scheduled task $T_{i'}$ instead of task T_i at time t_i.

iii) The processor is fully utilized during the interval $[t_i, t_d]$.

If the processor is ever idle in the interval $[t_i, t_d]$, then the analysis of Case 1 can be applied directly to the interval $[t_0, t_d]$ — where $t_0 > t_i + C_i$ is the end of the last idle period prior to time t_d — to reach a contradiction of condition (1).

Since $p_i > t_d - t_i$, fact (*i*) indicates that only tasks T_1 - T_i need be considered when computing d_{t_i,t_d}. Since the invocation of task T_i that is scheduled at time t_i has an initial deadline after time t_d, all task invocations occurring prior to time t_i with deadlines at or before t_d must have completed execution by time t_i and hence do not contribute to d_{t_i,t_d}. Similarly, since T_i has the last task invocation with initial deadline after t_d that executes prior to t_d, all invocations of tasks T_1 - T_{i-1} occurring prior to time t_d with deadlines after t_d, need not be considered. Lastly, since none of the invocations of tasks T_1 - T_{i-1} that are scheduled in the interval $[t_i, t_d]$ occurred at time t_i, the achievable demand due to tasks T_1 - T_{i-1} in the interval $[t_i, t_d]$ is the same as in the interval $[t_i+1, t_d]$. These observations, plus the fact the invocation of task T_i scheduled at time t_i must be completed before time t_d, indicate that the actual processor demand in $[t_i, t_d]$ is bounded by

$$d_{t_i,t_d} \leq d^*_{t_0,t_d} = C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_i+1)}{p_j} \right\rfloor C_j .$$

Let $L = t_d - t_i$. Substituting L into the above inequality yields

$$d_{t_i,t_d} \leq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor C_j .$$

Since (*iii*) indicates that there is no idle time in $[t_i, t_d]$, and since a task failed at time t_d, it follows that $d_{t_i,t_d} > t_d - t_i$ and hence $d_{t_i,t_d} > L$. Combining this with the inequality above yields

$$L < C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor C_j , \qquad (3.1)$$

Since the invocation of task T_i scheduled at time t_i has an execution deadline less than or equal to time t_d, it must be the case that $(t_i + 1) + P_{r_i} \leq t_d$. Hence:

$$t_d - (t_i + 1) \geq P_{r_i} ,$$
$$t_d - t_i > P_{r_i} ,$$
$$p_i > t_d - t_i > P_{r_i} ,$$
$$p_i > L > P_{r_i} .$$

Therefore inequality (3.1) above contradicts the assumption that condition (2) was true.

Case 2b: The invocation of task T_i scheduled at time t_i has an execution deadline greater than time t_d.

This will be the case if task T_i is either a non-resource requesting task ($r_i = 0$), or if $(t_i + 1) + P_{r_i} > t_d$. The implication of this case is that the invocation of task T_i scheduled at time t_i need not be completed before time t_d. That is, since the invocation of task T_i scheduled at time t_i has a deadline after t_d, it follows that T_i may be preempted by *any* task with an invocation interval contained within the interval $[t_i, t_d]$. This is possible because, since $t_d - t_i \leq P_{r_i}$, task T_i can not share a resource with any task that can possibly have an invocation interval contained within the interval $[t_i, t_d]$.

Let $t_0 > t_i$ be the later of the end of the last idle period in $[t_i, t_d]$ or the time task T_i last stops execution prior to t_d. Since the invocation of task T_i scheduled at time t_i has a deadline greater than t_d and since T_i is preemptable by any task that executes in $[t_i, t_d]$, all invocations of tasks occurring prior to time t_0 with deadlines less than or equal to t_d must have completed execution by t_0. The analysis of Case 1 can be applied directly to the interval $[t_0, t_d]$ to reach a contradiction of condition (1).

This concludes Case 2. We have shown that in all cases, if the EDF/DDM scheduling discipline fails, then either condition (1) or condition (2) from Theorem 3.2 must have been violated. This proves the theorem. □

Corollary 3.5: With respect to the class of scheduling algorithms that do not use inserted idle time, the EDF/DDM discipline is an optimal discipline for scheduling a set of sporadic tasks that share a set of serially reusable, single unit resources.

Proof: The proof follows immediately from Theorems 3.2 and 3.4. □

4. Multiple Phase Task Systems

In this section we demonstrate how the EDF/DDM scheduling algorithm developed for scheduling single phase sporadic tasks can be extended to successfully schedule multiple phase sporadic tasks that share a set of resources. The extension is straightforward and preserves the optimality of the EDF/DDM discipline.

4.1 Feasibility Conditions for Multiple Phase Task Systems

The following theorem gives the appropriate necessity conditions for the feasibility of a set of multiple phase tasks.

Theorem 4.1: Let τ be a set of multiple phase sporadic tasks

$$\{T_i = (s_i, \{(c_{ij}, C_{ij}, r_{ij}) \mid 1 \le j \le n_i\}, p_i) \mid 1 \le i \le n\},$$

sorted in non-decreasing order by period, that share a set of m serially reusable, single unit resources $R_1, R_2, ..., R_m$. If τ can be scheduled on a uniprocessor without inserted idle time, then:

1) $\sum_{i=1}^{n} \frac{E_i}{p_i} \le 1$,

2) $\forall i, 1 < i \le n$; $\forall k, 1 \le k \le n_i$ and $r_{ik} \ne 0$; $\forall L, P_{r_{ik}} < L < p_i - S_{ik}$:

$$L \ge C_{ik} + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor E_j ,$$

where:

- $E_j = \sum_{l=1}^{n_j} C_{jl}$,
- $P_{r_{ik}} = \underset{l \le j \le n}{\text{MIN}} (p_j \mid r_{jl} = r_{ik}$ for some l, $1 \le l \le n_j)$, and
- $S_{ik} = \begin{cases} 0 & \text{if } k = 1, \\ \sum_{j=1}^{k-1} c_{ij} & \text{if } 1 < k \le n_i. \end{cases}$

The feasibility conditions are similar to those for single phase tasks. The parameter E_i represents the maximum computational cost of an invocation of task T_i and replaces the C_i term in condition (1). Condition (2) now applies to only a resource requesting phase of task T_i rather than to the task as a whole. Because of this, the range of L in condition (2) is more restricted than in the single phase case. The range of L is more restricted because of the precedence constraints imposed on the execution of phases in multiple phase tasks. Since the k^{th} phase of a task T_i cannot start until all previous phases have terminated, the earliest time phase k can be scheduled is S_{ik} time units after the start of an invocation of T_i. Therefore, for the k^{th} phase of a task, the range of intervals of length L in which one must compute the achievable processor demand will be shorter than in the single phase case by the sum of the minimum costs of phases 1 through k–1. Also note that no demand due to phases of T_i other than k appear in (2). In the event that each task in τ consists of only a single phase, conditions (1) and (2) reduce to the conditions of Theorem 3.2.

Proof: To demonstrate the necessity of conditions (1) and (2) for arbitrary release times, by Lemma 3.1, it suffices to demonstrate the existence of release times for which conditions (1) and (2) are necessary for feasibility.

The construction for the necessity of condition (1) is identical to the one used in the proof of Theorem 3.2 and will not be repeated here. For condition (2) choose a task T_i, $1 < i \leq n$, and choose a phase k of T_i, $1 \leq k \leq n_i$, such that $r_{ik} \neq 0$, and $P_{r_{ik}} < p_i$. Let $s_i = 0$ and $s_j = S_{ik} + 1$ for all j, $1 \leq j \leq n$, $j \neq i$. This gives rise to the pattern of initial task invocations shown in Figure 4.1.

For all L, $L > P_{r_{ik}}$, the interval $[S_{ik}, S_{ik}+L]$ contains at least one entire invocation of a task that will require resource r_{ik}. Therefore, if τ is to be feasible then, in the worst case, the

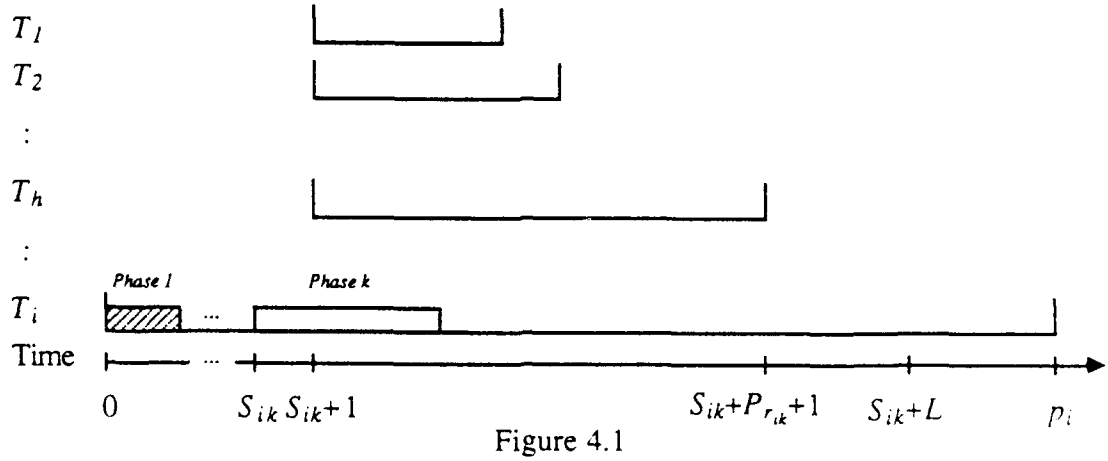


Figure 4.1

computation of task T_i started at time 0 must have its k^{th} phase completed in the interval $[S_{ik}, S_{ik}+L]$. Hence for all L, $P_{r_{ik}} < L < p_i - S_{ik}$, in the interval $[S_{ik}, S_{ik}+L]$, the achievable processor demand, $d^*_{S_{ik},S_{ik}+L}$, is

$$d^*_{S_{ik},S_{ik}+L} = C_{ik} + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor E_j .$$

For all L, $L > P_{r_{ik}}$, the interval $[S_{ik}, S_{ik}+L]$ contains at least one entire invocation of a task that will require resource r_{ik}. Therefore, if τ is to be feasible then, in the worst case, the computation of task T_i started at time 0 must have its k^{th} phase completed in the interval $[S_{ik}, S_{ik}+L]$. Hence for all L, $P_{r_{ik}} < L < p_i - S_{ik}$, in the interval $[S_{ik}, S_{ik}+L]$, the achievable processor demand, $d^*_{S_{ik},S_{ik}+L}$, is

$$d^*_{S_{ik},S_{ik}+L} = C_{ik} + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor E_j .$$

Note that it is not necessary for phases of task T_i beyond phase k to execute in $[0, L]$ in order to ensure that a task does not fail in the interval $[0, L]$. For τ to be feasible it must be case that $L \geq d^*_{S_{ik},S_{ik}+L}$, hence

$$L \geq C_{ik} + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor E_j . \qquad \square$$

4.2 Scheduling Multiple Phase Task Systems

The EDF/DDM scheduling discipline was originally formulated for single phase sporadic tasks. To see how it can be extended to handle tasks with multiple phases, it will be instructive to view a multiple phase sporadic task

$$T_i = (s_i, \{(c_{ij}, C_{ij}, r_{ij}) \mid 1 \leq j \leq n_i\}, p_i),$$

as set of n_i single phase sporadic tasks

$$\{T_{ij} = (s_i, (c_{ij}, C_{ij}, r_{ij}), p_i) \mid 1 \leq j \leq n_i\}.$$

For a given value of i, all tasks in $\{T_{ij} \mid 1 \leq j \leq n_i\}$ conceptually are invoked simultaneously and are scheduled such that the k^{th} invocation of task T_{ij}, $1 < j \leq n_i$, is not scheduled until the k^{th} invocation of task T_{ij-1} has completed execution. (Note that for a given value of i, since all tasks in $\{T_{ij} \mid 1 \leq j \leq n_i\}$ are invoked simultaneously, outstanding invocations of tasks T_{ij} will always have the same deadline. Therefore, the EDF/DDM scheduling discipline can be made to enforce the precedence constraints on the execution of these

single phase tasks by further biasing its algorithm for selecting a task for execution when there exist more than one ready task with the earliest deadline.) It should be clear that the execution of the set of single phase tasks $\{T_{ij}\}$ defined above will be equivalent to the execution of a multiple phase task T_i. This motivates the treatment of each phase of a multiple phase task as a logical single phase task. Specifically, each resource requesting phase of a multiple phase task should have both a contending and an execution deadline.

Let t_r be a point in time at which a task T_i is invoked. Let t_{sk} be the time that the k^{th} phase of the invocation of task T_i made at time t_r is first scheduled (commences execution) and let t_{ck} be the time that this phase terminates. In the interval $[t_r, t_{s1}]$, task T_i will have a *contending* deadline equal to $t_r + p_i$ as in traditional EDF scheduling. For all k, $1 \le k \le n_i$, if $r_{ik} \neq 0$ and $C_{ik} > 1$, then in the interval $[t_{sk}+1, t_{ck}-1]$, task T_i will have an execution deadline equal to $\text{MIN}(t_r + p_i, (t_{sk} + 1) + P_{r_{ik}})$. Between phases task T_i will be considered to be conceptually contending for the processor. At the time of the completion of each phase, t_{ck}, $1 \le k < n_i$, the deadline of task T_k will revert to the initial deadline for this invocation. Hence for all k, $1 \le k < n_i$, in the interval $[t_{ck}, t_{s(k+1)}]$, task T_i will have a deadline at time $t_r + p_i$. Figure 4.2 illustrates how a multiple phase task's deadline can change dynamically throughout an invocation interval. It shows an execution of a multiple phase task $T_i = (s_i, \{(3,3,r_{i1}), (3,3,r_{i2}), (10,10,r_{i3})\}, p_i)$ where each phase has an execution deadline that differs from its contending deadline.

We will refer to the extended version of the EDF/DDM scheduling discipline as the

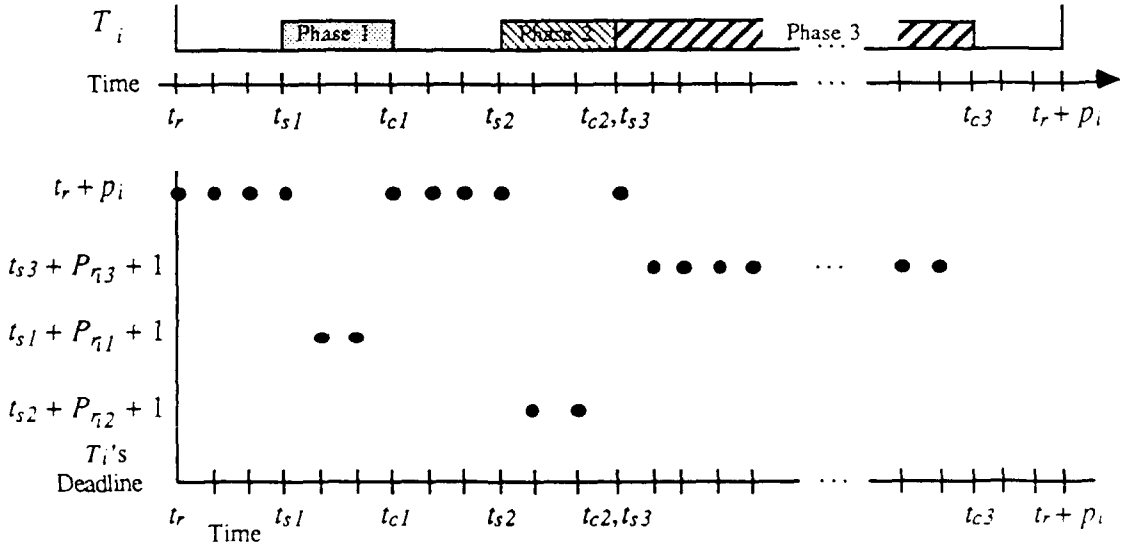


Figure 4.2

generalized EDF/DDM discipline. We again validate the design of this discipline by demonstrating that it is an optimal policy for scheduling a set of multiple phase tasks that share a set of resources. To prove optimality it suffices to show that the satisfaction of conditions (1) and (2) from Theorem 4.1 is sufficient for ensuring that the generalized EDF/DDM discipline will succeed in scheduling a set of multiple phase tasks with shared resources. To demonstrate that the discipline succeeds in scheduling a set of tasks it must be shown that (1) all invocations of all tasks complete execution before the end of their respective invocation intervals and that (2) the mutual exclusion constraints on the execution of resource requesting phases of tasks are respected. The following lemma demonstrates that the EDF/DDM scheduling discipline enforces the mutual exclusion constraints on the execution of resource requesting phases.

Lemma 4.2: The generalized EDF/DDM scheduling discipline ensures that resources are accessed in a mutually exclusive manner.

Proof: The proof is largely identical to the proof of Lemma 3.3 and will not be repeated here. □

Theorem 4.3: Let τ be a set of multiple phase sporadic tasks $\{T_1, T_2, ..., T_n\}$, sorted in non-decreasing order by period, that share a set of m serially reusable, single unit resources $R_1, R_2, ..., R_m$. The generalized EDF/DDM discipline will succeed in scheduling τ if conditions (1) and (2) of Theorem 4.1 hold.

Proof: It suffices to show that the use of the generalized EDF/DDM scheduling discipline guarantees that tasks will not fail if conditions (1) and (2) of Theorem 4.1 hold. The proof is quite similar to the proof of Theorem 3.4 and will be presented in an abbreviated manner.

Assume the contrary, *i.e.*, that conditions (1) and (2) hold and yet a task fails at some point in time when τ is scheduled by the generalized EDF/DDM algorithm. Let t_d be the earliest point in time at which a task fails. τ can be partitioned into three disjoint subsets A_1, A_2, A_3 as in the proof of Theorem 3.4. To bound the actual processor demand prior to t_d, it suffices to concentrate on the tasks in A_2. Let $b_1, b_2, ..., b_k$ be the invocation times immediately prior to t_d of the tasks in A_2. There are two main cases to consider.

Case 1: None of the invocations of tasks in A_2 occurring at times $b_1, b_2, ..., b_k$ are scheduled prior to time t_d.

This is identical to Case 1 in the proof of Theorem 3.2. If none of the invocations of tasks in A_2 occurring at times $b_1, b_2, ..., b_k$ are scheduled prior to t_d, then condition (1) could not have been true.

Case 2: Some of the invocations of tasks in A_2 occurring at times $b_1, b_2, ..., b_k$ are scheduled prior to time t_d.

Let T_i be the last task in A_2 to execute prior to t_d. Let h be the last phase of task T_i to execute prior to time t_d. Let t_i be the point in time at which phase h of task T_i commences execution (is scheduled for the first time). There are two sub-cases to consider depending on whether or not the phase of task T_i scheduled at time t_i has an execution deadline less than or equal to time t_d.

Case 2a: The phase of task T_i scheduled at time t_i has an execution deadline less than or equal to time t_d.

If a task fails at time t_d then facts (*i*) - (*iii*) from the proof of Theorem 3.4 hold for the present case. The actual processor demand in $[t_i, t_d]$ is bounded by

$$d_{t_i,t_d} \leq C_{ih} + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_i+1)}{p_j} \right\rfloor E_j .$$

(Note that since phase h of task T_i is the last phase of T_i to execute prior to t_d, T_i contributes only C_{ih} to d_{t_i,t_d})

Let $L = t_d - t_i$. Since there can be no idle time in $[t_i, t_d]$, and since a task failed at t_d, it follows that $d_{t_i,t_d} > t_d - t_i$, and hence $d_{t_i,t_d} > L$. Therefore,

$$L < C_{ih} + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor E_j ,$$

The earliest phase h of task T_i can be scheduled is S_{ih} time units after T_i is invoked. Therefore, since T_i was in A_2, it follows that $p_i - S_{ih} > t_d - t_i$. Moreover, since phase h of task T_i had an execution deadline less than or equal to t_d, we have $t_d - t_i > P_{r_i}$ and hence $p_i - S_{ih} > L > P_{r_i}$. Therefore the above inequality contradicts the fact that condition (2) was assumed to be true.

Case 2b: The phase of task T_i scheduled at time t_i has an execution deadline greater than time t_d.

Let $t_0 > t_i$ be the later of the end of the last idle period in $[t_i, t_d]$ or the time the h^{th} phase of task T_i last stops execution prior to t_d. The analysis of Case 1 can be applied directly to the interval $[t_0, t_d]$ to reach a contradiction of condition (1).

This concludes Case 2. We have shown that in all cases, if the generalized EDF/DDM scheduling algorithm fails, then either condition (1) or condition (2) of Theorem 4.1 must have been violated. This proves the theorem. □

Theorem 4.4: With respect to the class of scheduling algorithms that do not use inserted idle time, the generalized EDF/DDM discipline is an optimal discipline for scheduling a set of multiple phase sporadic tasks that share a set of serially reusable, single unit resources.

Proof: The proof follows immediately from Theorems 4.1 and 4.3. □

5. Discussion

In this section we present an $O(mp_n)$ algorithm for deciding if a set of tasks is feasible where p_n is the period of the "largest" task and m is the number of shared resources in the system. In addition we revisit some of the assumptions and restrictions present in the system model of Section 2. Having proved necessary and sufficient conditions for feasibility, we can provide additional motivation for our specific choice of tasking and resource model, our emphasis on scheduling without inserted idle time, and the necessity of a pessimistic scheduling discipline. Lastly, we discuss some issues concerning the implementation of our system model.

5.1 The Complexity of Deciding Feasibility

Conditions (1) and (2) of Theorem 4.1 can be used as the basis of a decision procedure for deciding the feasibility of a set of sporadic tasks that share a set of serially reusable, single unit resources. By Theorems 4.3 and 4.4, a set of tasks will be feasible if and only if they satisfy conditions (1) and (2). A set of tasks can be described with $O(\sum_{i=1}^{n} n_i)$ inputs. Deciding if condition (1) holds is straightforward and can be performed in time linear in the number of inputs. As described next, we can determine if a set of sporadic tasks satisfy condition (2) in time $O(mp_n)$. Note that for all tasks T_i, the maximum computational cost of the task is at least as big as the number of phases in the task, *i.e.*, $E_i \geq n_i$. Therefore, for task sets of size n that satisfy condition (1)

$$p_n \geq \sum_{i=1}^{n} E_i \geq \sum_{i=1}^{n} n_i.$$

A set of sporadic tasks can be tested against condition (2) in time $O(mp_n)$, as follows. Let

$$f(L) = \sum_{j=1}^{n} \left\lfloor \frac{L-1}{p_j} \right\rfloor E_j ,$$

Intuitively, $f(L)$ is the achievable processor demand in the interval $[0, L-1]$ when all tasks are released at time zero. To test condition (2) we restrict our attention to values of L between p_1 and p_n. To compute $f(L)$ for all L, $p_1 < L < p_n$, initialize an array of integers A of size p_n to zero. For each task T_k, $1 \leq k \leq n$, add E_k to location j of array A for all j that are multiples of p_k. At the completion of this process the sum of the first $l-1$ locations of A will be $f(l)$.[5] Using this method, the total time required for the computation of $f(L)$ for all L, $p_1 < L < p_n$, is $O(p_n)$ plus the time required to compute the E_i plus the maximum number of task invocations that must complete execution before time $p_n - 1$ when all tasks are released at time zero. If a set of tasks satisfies condition (1) then the second and third terms can be at most p_n. Therefore the time required to compute $f(L)$ for all L, $p_1 < L < p_n$, is $O(p_n)$. Note that if $l < p_i$, for some task T_i, then

$$f(l) = \sum_{j=1}^{i-1} \left\lfloor \frac{l-1}{p_j} \right\rfloor E_j .$$

For each shared resource R, let

$$M_R(p) = \underset{P_R < L < p}{\text{MIN}} \left(L - f(L) \right).$$

Intuitively, $M_R(p)$ is the minimum amount of time the processor will have been idle in the interval $[0, L-1]$, for all $L < p$, if all tasks with periods less than p are released at time zero (and all tasks with period greater than or equal to p are released at or after time p). For a resource R, the time required for computing $M_R(p)$ for all p, $P_R < p < p_n$, is $O(p_n)$. If there are m resources in the system then the total time required to compute $M_R(p)$ for all resources in the system is $O(mp_n)$. A set of tasks will satisfy condition (2) if and only if for each task i, $1 < i \leq n$, and each phase k, $1 \leq k \leq n_i$, $M_{r_{ik}}(p_i - S_{ik}) \geq C_{ik}$. Given $M_R(p)$ for all R and for all p, $P_R < p < p_n$, this final determination can be made in time $O(\sum_{i=1}^{n} n_i)$.

---

[5] Note that the array A need only be of size $p_n - p_1$ since for all l, $0 \leq l < p_1$, $f(l) = 0$. However, this optimization does not effect the time complexity of the computation.

Therefore, the time required to the decide feasibility of a set of sporadic tasks is dominated by the time required to compute $M_R(p)$ for all resources R; namely $O(mp_n)$. Variations of this algorithm are discussed in [Jeffay et al. 90].

Note that the time complexity depends on the value of one of the inputs. Since the size of an input cannot be expressed as a polynomial in the length of the input, our decision procedure is a pseudo-polynomial time algorithm [Garey & Johnson 79]. However, this does not necessarily imply intractability in practice. For any bound on the size of the inputs, our algorithm is polynomial in this bound. Therefore, if we impose an upper bound on the size of the inputs, say 2^{16}, then the decision procedure is polynomial for these restricted problems. For descriptions of task sets that are most likely to be encountered in practice, one can efficiently determine the feasibility of the tasks.

Parameters for the decision procedure such as task periods are typically specified as part of the system design or are derivable from an examination of the execution environment. Minimum and maximum phase execution times can be computed by hand or by automated tools. For example, a compiler that emits minimum and maximum execution times for source language level constructs (*e.g.*, procedures, statements, expressions) has been reported by Park and Shaw [Park & Shaw 90].

5.2 Optimistic Versus Pessimistic Scheduling

In Section 2 we derived the DDM rule for dynamically modifying the deadline for an invocation of a resource requesting task. This rule was introduced to ensure that blocked tasks become unblocked (ready) as soon as possible. The DDM rule is pessimistic in the sense that it requires *all* invocations of resource requesting tasks to execute with a potentially modified deadline independently of whether or not any blocking has, or will, actually occur. Furthermore, if an invocation of a resource requesting task is assigned a new deadline after commencing execution, the DDM rule is pessimistic in the choice of the new deadline. By assigning a new deadline that is a function of P_j, for the appropriate value of j, the DDM rule is in effect assuming that if a task will become blocked it will be the smallest task that shares a resource with the blocking task. Although it was demonstrated that the DDM rule lead to an optimal scheduling discipline, it is instructive to examine some of the pitfalls of a more optimistic scheduling strategy.

A more optimistic approach to scheduling would be to modify the deadline of a resource requesting task only when an invocation of the task actually blocks some other task and in

addition have the new deadline be a function of the period of the blocked task (*e.g.*, see [Jeffay 89b]). For example, in the single phase case, when a resource requesting task T_i is invoked at time t_r, the invocation should have an initial deadline equal to $t + p_i$ as before. Let t_s be the time this invocation of task T_i commences execution and let t_c be the time it terminates. If at time t', $t_s < t' < t_c$, an invocation of some other task T_j becomes blocked by T_i, then at time t' the deadline of T_i should be advanced to time MIN($t_r + p_i$, $t' + p_j$). For example, consider the following (feasible) set of single phase tasks: (recall T = (*release time*, (*minimum cost*, *maximum cost*, *resource*), *period*))

$$\tau = \{\ T_1 = (2, (1, 1, 1), 3)$$
$$T_2 = (1, (2, 2, 0), 7)$$
$$T_3 = (0, (3, 3, 1), 10)\ \}.$$

Figure 5.1 shows the execution of these tasks under the optimistic scheduling policy outlined above. (For comparison, Figure 5.2 shows the execution of the tasks under the EDF/DDM scheduling discipline.) Initially task T_3 is scheduled. At time 1 the EDF/DDM policy would assign task T_3 an execution deadline of time $1 + P_1 = 4$. Under the proposed optimistic policy, since no task is actually blocked at time 1, task T_3 would retain its original deadline of time 10. Therefore, since task T_2 is initially invoked at time 1 and has a deadline at time 8, the optimistic policy would allow task T_2 to preempt task T_3 at time 1 since T_2 has the earliest deadline at time 1. Task T_1 is immediately blocked when it is invoked at time 2 since it requires the resource held by task T_3. At this point the optimistic policy would assign task T_3 a deadline of time $2 + p_1 = 5$; enabling T_3 to preempt the currently executing task. This results in the initial invocations of all tasks to complete execution before their deadlines.

This example suggests that the optimistic policy we have proposed is at least as good as the pessimistic EDF/DDM policy. It turns out that this is not the case. To see that the pessimistic deadline modification rule is indeed warranted, consider the following set of

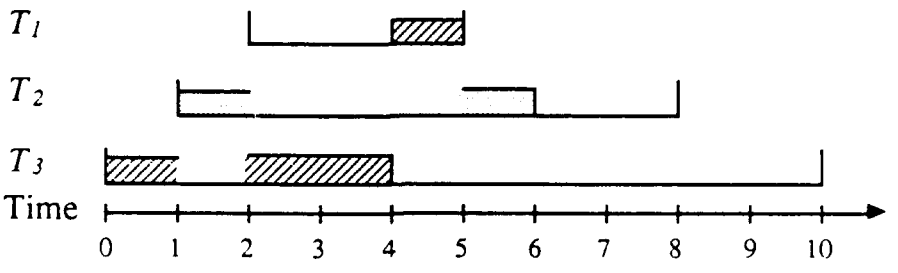


Figure 5.1

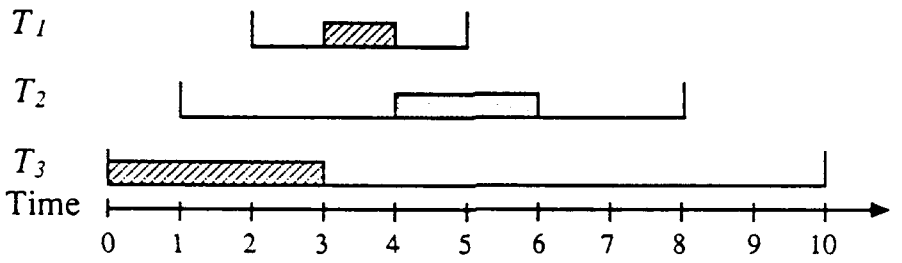


Figure 5.2

single phase tasks that share resource R_1 and R_2:

$$\tau = \{ \; T_1 = (3, (1, 1, 1), 4) \\ T_2 = (2, (2, 2, 2), 6) \\ T_3 = (1, (3, 3, 1), 15) \\ T_4 = (0, (3, 3, 2), 17)\}.$$

Figure 5.3 depicts a simulation of an EDF scheduling discipline with the optimistic deadline modification strategy. When tasks T_4 and T_3 are invoked they will have deadlines at time 17 and 16 respectively. Task T_4 will execute until time one at which point it will be preempted and task T_3 will be scheduled. At time 2 an invocation of task T_2 has the nearest deadline but is blocked by the uncompleted invocation of task T_4. Therefore, at time 2 task T_4 is assigned a new deadline of time $2 + p_2 = 8$. This causes task T_4 to resume execution at time 2. Similarly, since task T_1 becomes blocked by task T_3 at time 3, at time 3 the invocation of task T_3 made at time 1 is assigned a new deadline of time $3 + p_1 = 7$ at time 3. This causes task T_3 to resume execution at time 3. Eventually task T_2 misses a deadline at time 8.

However, this task set is feasible as it satisfies both conditions (1) and (2) of Theorem 3.2. Figure 5.4 shows the effect of scheduling this task set according to the EDF/DDM

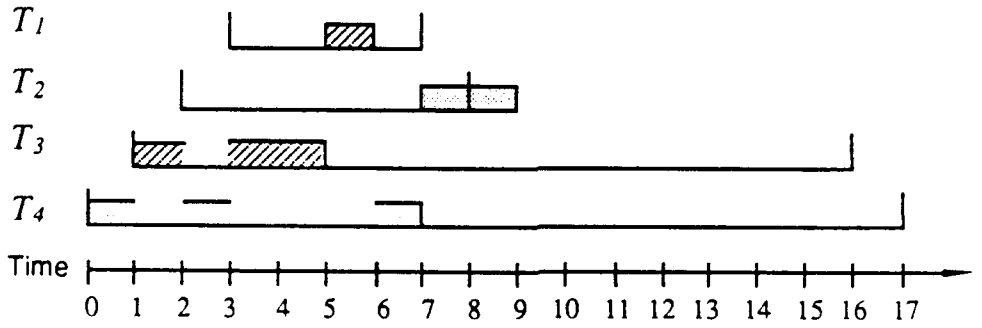


Figure 5.3

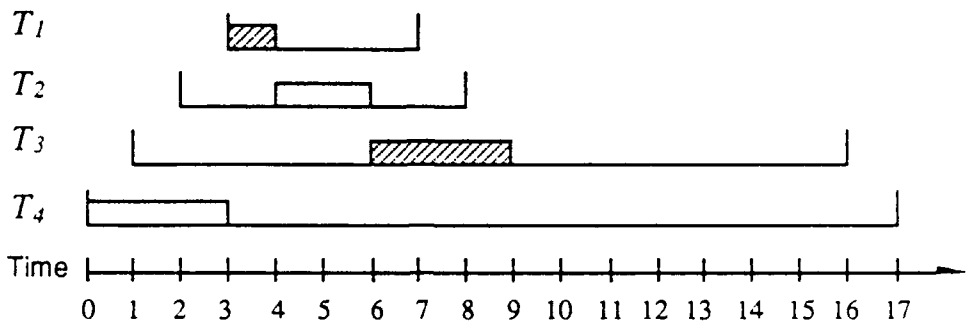


Figure 5.4

discipline. Note that the execution of these tasks is non-preemptive. The reason for the failure of the more optimistic policy can be seen by revisiting the construction used in the proof of Theorem 3.2 to demonstrate the necessity of condition (2) for the feasibility of a set of single phase tasks. Condition (2) describes a least upper bound on the achievable processor demand for an interval I of length L that is contained within the invocation interval of a resource requesting task T_i. The key observation is that this bound contains the computational cost of only a *single* invocation of a single task (namely task T_i) that can not be wholly contained within the interval I (see Figure 3.2). That is, in the worst case there exists only one task invocation not contained within the interval I that *must* be completed within I. Theorem 3.4 has shown that this scenario is indeed the worst case one need consider.

The optimistic deadline modification strategy outlined above is inferior because it admits the computational cost of more than one task with an invocation interval not wholly contained within the interval I into the processor demand for this interval. In the example above, consider the interval $I = [0,8]$ contained within an invocation interval of the resource requesting task T_4. Under an optimistic deadline modification strategy, invocations of both tasks T_3 and T_4 *must* be completed within the interval I. Under the pessimistic deadline modification strategy only an invocation of task T_4 must be completed within the interval I. For this interval of length $L = 8$, the processor demand is higher under the optimistic deadline modification strategy than under the pessimistic strategy. Any scheduling policy that allows the processor demand within the invocation interval of a task, to exceed the bound given by condition (2) will necessarily be non-optimal. The optimistic deadline modification strategy fails in the second example for precisely this reason. Therefore, although the EDF/DDM discipline always schedules resource requesting tasks as if the

smallest competing task becomes blocked immediately after any resource requesting task commences execution, such an approach is indeed necessary.

5.3 Feasibility Versus Processor Utilization

Condition (1) of Theorems 3.2 and 4.1 requires that the cumulative utilization of a set of tasks not overload the processor. It is important to note that this is the only feasibility condition that constrains the achievable utilization of a real-time task set. Although condition (2) of these theorems constrains the achievable utilization over a relatively short and well-defined set of intervals, it does not constrain the overall processor utilization. The feasibility of a set of sporadic tasks that share a set of resources is not a function of processor utilization (to the extent that the tasks do not overload the processor). It is possible to conceive of both *feasible* task sets that have a processor utilization of 1.0, and *infeasible* task sets that have arbitrarily small processor utilization.

The implication of this is that manipulating infeasible task sets according to such "rules-of-thumb" as lowering the overall processor utilization will not necessarily yield a feasible task set. For example, one approach to scheduling tasks that share resources has been to reduce the analysis of a set of periodic tasks with preemption or mutual exclusion constraints to the analysis of a set of periodic tasks without such constraints [Mok et al. 87, Sha et al. 90]. In this manner, the results developed for independent periodic tasks can be applied. For periodic tasks with no preemption constraints, the conditions that are necessary and sufficient for guaranteeing response times are stated in terms of the processor utilization of the system. Tasks with no preemption constraints can be scheduled if

$$U = \sum_{i=1}^{n} \frac{E_i}{p_i} \leq \alpha,$$

where the value of α, $0 < \alpha \leq 1$, varies according to the problem statement [Liu & Layland 73]. For our purposes we can consider α to be a constant. (In our analysis we had $\alpha = 1$.) The reductions from the constrained task system to the independent task system typically impose further restrictions on the utilization of the system. A common form for the schedulability conditions for task sets with preemption constraints is $U \leq \alpha - B$, where B is a function of the durations for which tasks in the system can be blocked [Leinbaugh 80, Stoyenko 87, Mok et al. 87, Sha et al. 90]. The reduction process results in conditions that are sufficient for ensuring the correctness of a set of tasks but that are not necessary. In

effect, these methods are sacrificing processor utilization to gain schedulability. Our work demonstrates that, in principle, one need not make such a trade-off.

5.4 Scheduling Periodic Tasks and Scheduling With Inserted Idle Time

Although we have focused on modeling a real-time systems as a set of sporadic tasks, a more common approach is to view a real-time systems as a set of *periodic* tasks [Mok 83]. A periodic task is the special case of a sporadic task obtained when a sporadic task is invoked every p time units after it is released (where p is the period of the sporadic task). The conditions sufficient for a set of sporadic tasks to be feasible will therefore be sufficient conditions for ensuring the feasibility of a set of periodic tasks. Indeed the generalized EDF/DDM scheduling discipline will correctly schedule a set of periodic tasks that share a set of serially reusable, single unit resources if the conditions of Theorem 4.1 hold. These conditions are, however, not necessary for the feasibility of a set of periodic tasks. That is, the generalized EDF/DDM scheduling discipline is not an optimal algorithm for scheduling periodic tasks that share resources. For the simplest form of mutual exclusion constraints (*i.e.*, a non-preemptive system), the problem of determining necessary conditions for the feasibility of a set of periodic tasks with arbitrary release times is known to be NP-hard in the strong sense [Jeffay at al. 90]. Moreover, if an optimal algorithm exists for the non-preemptive scheduling of periodic tasks then P = NP [Jeffay et al. 90]. It is for these reasons that we have limited our attention to sporadic tasks.

The intractability of deciding feasibility for a set of periodic tasks arises from our inability to efficiently determine if the processor demand given in condition (2) can ever actually occur. That is, for a set of periodic tasks, one cannot efficiently determine if there can exist an interleaving of task invocations such that there exists an interval of length L in which the processor demand is given by condition (2). The optimality of the results in this paper are primarily due to the non-determinism allowed in the behavior of a sporadic task. Since there may exist an arbitrarily long delay between invocations of sporadic tasks, one can argue that there can always exist an interval of length L in which a set of sporadic tasks will realize the processor demand given in condition (2).

The non-determinism in the behavior of sporadic tasks is also responsible for consideration of on-line scheduling policies. It will not be possible to generate a schedule off-line if the invocation times of all tasks are unknown. For similar reasons, we have largely ignored the investigation of scheduling policies that use inserted idle time. In order for inserted idle

time to function correctly, it would seem to require that the scheduler know when tasks will next be invoked. In general, this will not be possible for sporadic tasks.

5.5 Implementation Considerations

The tasking model considered in this paper can be efficiently implemented if the EDF/DDM scheduling discipline is employed. This is primarily due to a property of priority driven schedulers. If there are no shared resources in a system then tasks may, in principle, preempt one another at arbitrary points. In particular, when such a set of tasks are scheduled by an EDF scheduling discipline, the schedule produced has the property that if an invocation of a task is preempted at some time t_p and resumed at some later time t_r, all tasks that execute in the interval $[t_p, t_r]$ execute to completion.[6] When EDF scheduling is used, this suggests a possible implementation strategy wherein all tasks share a single run-time stack.[7] This implementation strategy for a real-time tasking model has been called *featherweight tasks* [Baker 90a]. The use of a single stack can greatly improve memory utilization as well as lower the cost of dispatching and preempting tasks. Although the EDF/DDM scheduling discipline dynamically changes the deadline of resource requesting tasks, it preserves the ability of a set of tasks to be implemented using a single stack.

In order to apply the feasibility conditions of Theorem 4.1 in practice, one must account for the overhead of an implementation of an EDF/DDM scheduler. Throughout this paper we have ignored the cost of selecting, dispatching, and preempting a task. If the scheduling priority of tasks changes over time, as is the case in EDF/DDM scheduling, one of the most difficult implementation costs to appropriately quantify is the cost of preempting a task. It would therefore be useful to determine, for a given set, if allowing preemption between tasks is indeed necessary for feasibility. By combining individual resources into resource classes, one can represent a task system with m shared resources, as a system with k shared resources, for $1 \leq k \leq m$. (In the context of a concurrent program this amounts to using a single monitor for accessing a set of resources.) In this manner we can, roughly speaking, identify the "minimum" number of logical resources necessary for ensuring the schedulability of a set of tasks. For example, when using an EDF/DDM scheduler, if there exist two resources R_i and R_j, $i \neq j$, such that $P_i = P_j$, then a resource R_i requesting task

---

[6] Assume the EDF scheduler breaks ties according to a static priority assignment to tasks.

[7] This assumes a "lightweight" task implementation wherein all tasks execute within the same address space.

will never preempt a resource R_j task (nor execute while such a task is preempted) and vice versa. Therefore, if $P_i = P_j$, one can always treat resources R_i and R_j as a single logical resource .

For a given set of resources, there is an exponential number of possible resource classes to consider. However in practice the number of resources in a system is likely to be small and the process of enumerating and testing the feasibility of the various modified problem statements may be performed off-line.

Even if the number of logical resources required for feasibility is close to the number of actual resources in the system, we believe that in practice the number of tasks that are able to preempt other tasks will be small. For example, note that in each (admittedly contrived) example in this paper, the schedules produced by the EDF/DDM scheduling discipline have been *non-preemptive*. This is not by accident. In the case of single phase tasks, if $P_i \leq P_j$ then no resource R_j requesting task can ever preempt a resource R_i requesting task. This implies that there will always exist a group of tasks that may never preempt any resource requesting task. Furthermore, since a task T_k may preempt a resource R_i requesting task only if $p_k < P_i$, T_k can either preempt every resource R_i requesting task or it cannot preempt any such task. Based on these observations and our experience with applying the EDF/DDM discipline to actual task sets, we conjecture that if preemption among tasks is required for feasibility, it will be limited to a few tasks. For these tasks one may account for the cost of preemption by inflating their cost parameter c to include the cost of preempting a task. Further experience with constructing systems according to the model of Section 2 is clearly needed.

5.6 Other Paradigms of Resource Usage

Throughout this work we have assumed that tasks require at most one resource per phase and that phases are statically ordered. The latter restriction can be mitigated to a limited extent by judicious use of minimum phase execution time cost parameter c. A zero value for the minimum cost can be used to model simple branching logic that controls the order of phase execution. An alternate approach described by Stoyenko is to explicitly test the feasibility of all possible interleavings of task invocations for all possible phase orderings [Stoyenko 87]. We have chosen to restrict the programming model in order to ensure a simple test for feasibility.

The restriction that phases require at most one resource is certainly unrealistic for real-time systems such as in transaction systems where phases may require multiple resources

simultaneously. Recall that the initial motivation for our consideration of a single resource per phase arose from the use of monitors in concurrent programming languages. In this context we have operationally defined a resource as a monitor. The use of multiple resources simultaneously by a task corresponds to the "nested monitor problem" in the concurrent programming literature [Haddon 77, Lister 77]. Largely because of the problems associated with deadlock, many popular concurrent programming languages such as Modula[8], Mesa, and Concurrent Euclid do not allow nested monitor calls [Wirth 77, Lampson & Redell 80, Holt 83]. We have therefore not been motivated to consider phases that require multiple resources simultaneously. From a pragmatic standpoint, if in practice it is the case that the number of tasks that can preempt one another is indeed small, as conjectured in Section 5.3, then we would argue that there is little to be gained by investigating more complex models of shared resources. It would be better to simply consider the resources that a phase requires simultaneously as a single logical resource. This reduces the problem to the one considered in this paper.

From our perspective, a more interesting model to study is one that relaxes the mutual exclusion constraints on the access to resources. In this work resources have been required to be accessed in a mutually exclusive manner. Other models of models of exclusion, such as readers/writers, warrant consideration. We plan to investigate such problems in the future.

6. Summary and Conclusions

We have presented a model of a real-time system consisting of a set of sporadic tasks that share a set of serially reusable, single unit resources. Sporadic tasks are a generalization of periodic tasks and are well-suited for representing event driven processes. Tasks are composed of a sequence of phases. Each phase is a contiguous sequence of statements that possibly require exclusive access to a resource. Resources are shared software objects, such as data structures. Our treatment of resources has been motivated by the use of monitors in contemporary concurrent programming languages.

For an arbitrary instance of the model the goal is to determine if it is possible to schedule the tasks on a single processor such that:

---

[8] Modula allows lexically nested monitors, however, this is compatible with our one resource per phase paradigm.

- no task fails (every invocation of every task completes execution at or before the end of its invocation interval) and
- each instance of each resource requesting phase has exclusive access to the resource it requires for the duration of the phase.

We have identified conditions that are both necessary and sufficient for scheduling a set of tasks without the use of inserted idle time. Moreover, with respect to the class of algorithms that do not use inserted idle time, we have developed an optimal algorithm for scheduling sporadic tasks that share resources. This algorithm, called the *earliest deadline first with dynamic deadline modification* (EDF/DDM) algorithm, is an extension to the well-known EDF algorithm. Under an EDF/DDM scheduler, tasks that require exclusive access to resources have two types of deadlines: a *contending* deadline for the initial acquisition of the processor, and an *execution* deadline for subsequent execution. The EDF/DDM policy ensure that tasks that become blocked due to mutual exclusion constraints are resumed as soon as possible. This policy is pessimistic in the sense that it always assumes the act of scheduling a resource requesting task will result in a competing task becoming blocked. Our analysis has demonstrated that this pessimistic approach is warranted.

7. Acknowledgments

We would like to thank Peter Calingaert, Dan Poirier, Don Stanat, and Don Stone for their comments on earlier drafts of this paper. This work was supported in part by a Digital Faculty Program grant from Digital Equipment Corporation.

8. References

[Baker 90a] Baker, T.P., *A Stack-Based Resource Allocation Policy for Realtime Processes*, Proc. Eleventh IEEE Real-Time Systems Symp., Lake Buena Vista, FL, December 1990, pp. 191-200.

[Baker 90b] Baker, T.P., *Preemption vs. Priority, and the Importance of Early Blocking*, Proc. Seventh IEEE Workshop on Real-Time Operating Systems and Software, Charlottesville, VA, May 1990, pp. 44-48.

[Chen & Lin 90] Chen, M.-I., Lin, K.-J., *Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems*, **Real-Time Systems**, Vol. 2, No. 4, (November 1990), pp. 325-346.

[Conway et al. 67] Conway, R.W., Maxwell, W.L., Miller, L.W., **Theory of Scheduling**, Addison-Wesley, Reading, MA, 1967.

[Garey et al. 81] Garey, M.R., Johnson, D.S., Simons, B.B., and Tarjan, R.E., *Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines*, **SIAM J. Computing**, Vol. 10, No. 2, (May 1981), pp. 256-269.

[Jeffay 89a] Jeffay, K., *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*, Ph.D. Thesis, University of Washington, Department of Computer Science, Technical Report #89-09-15, September 1989.

[Jeffay 89b] Jeffay, K., *Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints*, Proc. Tenth IEEE Real-Time Systems Symp., Santa Monica, CA, December 1989, pp. 295-305.

[Jeffay et al. 90] Jeffay, K., Anderson, R., Martel, C.U., *On Optimal, Non-Preemptive Scheduling of Periodic and Sporadic Tasks,* University of N. Carolina at Chapel Hill, Department of Computer Science, Technical Report TR90-019, March 1990. (Submitted for publication.)

[Haddon 77] Haddon, B.K., *Nested Monitor Calls*, **ACM Operating Systems Review**, Vol. 11, No. 4, (October 1977), pp. 18-23.

[Hoare 74] Hoare, C.A.R., *Monitors: An Operating System Structuring Concept*, **Comm. of the ACM**, Vol. 17, No. 10, (October 1974), pp. 549-557.

[Holt 83] Holt, R.C., **Concurrent Euclid, The UNIX System, and TUNIS**, Addison-Wesley, Reading, MA, 1983.

[Kligerman & Stoyenko 86] Kligerman, E., Stoyenko, A.D., *Real-Time Euclid: A Language for Reliable Real-Time Systems*, **IEEE Trans on Soft. Eng.**, Vol. SE-12, No. 9, (September 1986), pp. 941-949.

[Lampson & Redell 80] Lampson, B.W., Redell, D.D., *Experience with Processes and Monitors in Mesa*, **Comm. of the ACM**, Vol. 23, No. 2, (February 1980), pp. 105 - 117.

[Leinbaugh 80] Leinbaugh, D.W., *Guaranteed Response Times in a Hard-Real-Time Environment*, **IEEE Trans. on Soft. Eng.**, Vol. SE-6, No. 1, (January 1980), pp. 85-91.

[Lister 77] Lister, A., *The Problem of Nested Monitor Calls*, **ACM Operating Systems Review**, Vol. 11, No. 3, (July 1977), pp. 5-7.

[Liu & Layland 73] Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, **Journal of the ACM**, Vol. 20, No. 1, (January 1973), pp. 46-61.

[Mok 83] Mok, A.K.-L., *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, MIT, Department of EE and CS, MIT/LCS/TR-297, May 1983.

[Mok et al. 87] Mok, A.K.-L., Amerasinghe, P., Chen, M., Sutanthavibul, S., Tantisirivat, K., *Synthesis of a Real-Time Message Processing System with Data-driven Timing Constraints*, Proc. Eighth IEEE Real-Time Systems Symp., San Jose, CA, December 1987, pp. 133 - 143.

[Park & Shaw 90] Park, C.Y., Shaw, A.C., *Experiments With a Program Timing Tool Based On Source-Level Timing Schema*, Proc. Eleventh IEEE Real-Time Systems Symp., Lake Buena Vista , FL, December 1990, pp. 72-81. (To appear: special issue IEEE Transactions on Computers.)

[Sha et al. 90] Sha, L., Rajkumar, R., Lehoczky, J.P., *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, **IEEE Trans. on Computers**. Vol. 39, No. 9, (September 1990), pp. 1175-1185.

[Stoyenko 87] Stoyenko, A.D., *A Schedulability Analyzer for Real-Time Euclid*, Proc. Eighth IEEE Real-Time Systems Symp., San Jose, CA, December 1987, pp. 218 - 227.

[Wirth 77] Wirth, N., *Modula: A Language for Modular Multiprogramming*, **Software Practice and Experience**, Vol. 7, No. 1, (January 1977), pp. 3-35.

[Xu & Parnas 90] Xu, J., Parnas, D.L., *Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations*, **IEEE Trans. on Soft. Eng.**, Vol SE-16, No. 3, (March 1990), pp. 360-369.

[Zhao et al. 87a] Zhao, W., Ramamritham, K., Stankovic, J.A., *Scheduling Tasks with Resource Requirements in Hard Real-Time Systems*, **IEEE Trans. on Soft. Eng.**, Vol. SE-13, No. 5, (May 1987), pp. 564-577.

[Zhao et al. 87b] Zhao, W., Ramamritham, K., Stankovic, J.A., *Preemptive Scheduling Under Time and Resource Constraints*, **IEEE Trans. on Computers**, Vol. C-36, No. 8, (August 1987), pp. 949-960.