Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints

Kevin Jeffay*
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

Abstract: This paper examines the problem of guaranteeing response times, on a uniprocessor, to sporadic tasks with preemption constraints. The preemption constraints arise from the fact that tasks require exclusive access to shared software resources during portions of their computations. The primary objective is to determine conditions under which we can guarantee a response time to each task which is less than or equal to the task's minimum inter-execution request time. We analyze three different characterizations of a task's resource requirements. We show that for restricted patterns of resource usage, there exist synchronization and scheduling disciplines which are optimal for executing these tasks.

1. Introduction

Real-time systems differ from more traditional multiprogramming systems in that real-time systems have a dual notion of correctness. Besides 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. To assess the temporal correctness of a real-time system, we wish to determine upper bounds on the response times of the tasks in the the system. These bounds will be determined analytically so that rigorous guarantees of temporal correctness can be made. For our purposes, a real-time system is a set of sporadic tasks that share a set of software resources to accomplish a particular objective in real-time. A sporadic task is a task which makes repeated requests for execution with a lower bound on the interval between requests. This paper examines the problem of guaranteeing response times to real-time tasks that require exclusive access to software resources on a uniprocessor. We wish to guarantee a response time to each task that is less than or equal to the task's minimum inter-request time. This work is part of a larger design system for hard-real-time systems [Jeffay 89].

To guarantee response times to tasks, one must have information concerning the implementation of a real-time system. At a minimum, we must know how tasks synchronize access to shared resources and how tasks are scheduled. We propose a programming discipline for implementing real-time tasks where access to each shared resources is controlled by a monitor with `WAIT` and `BROADCAST` synchronization primitives [Hoare 74, Lampson & Redell 80]. For task scheduling, we further propose that tasks be selected for execution according to an *earliest deadline first* selection rule [Liu & Layland 73]. We will show that for certain characterizations of a task's resource requirements, these disciplines can, in principle, result in an *implementation strategy* for constructing real-time systems which will be able to guarantee response times to tasks whenever it is possible to do so. In this sense, we will claim this combination of synchronization and scheduling disciplines is optimal for guaranteeing response times to sporadic tasks with preemption constraints.

In the following section we present an abstract description of a real-time system in terms of a set of sporadic tasks. Section three presents an implementation strategy for constructing a real-time system. We present the synchronization primitives we will use and describe our scheduling policy. The following three sections consider three characterizations of a task's resource requirements. Section four considers a set of sporadic tasks that share a single resource. We show that our implementation strategy is optimal for this task characterization. Section five extends the analysis to include sporadic tasks which share a set of resources, but where each task requires only a single resource. We show that under restricted conditions our implementation strategy is again optimal. Section six extends the analysis to encompass the most general task system. This is a set of sporadic tasks which share a set of resources and where each task may require multiple resources. We show that the results of Section five are sufficient for analyzing such a task system. Section seven discusses our results, reviews some related work, and outlines the contributions of this work.

2. Tasking Model

We first present an abstract description of a sporadic task. This description will be useful for establishing conditions which, independent of an implementation strategy, are necessary for guaranteeing response times to individual tasks.

A real-time system τ, is a set of *sporadic* tasks. A *sporadic* task T is a 3-tuple (s, c, p) where

- s = start or release time: the time of the first request for execution of task T,
- c = computational cost: the time to execute task T to completion on a dedicated uniprocessor, and
- p = "period": a lower bound on the interval between requests for execution of task T.

* Supported in part by a Graduate Fellowship from the IBM Corporation, and in part by a grant from the National Science Foundation (number CCR-8700435).

Throughout this paper we consider a discrete time model. In this domain we assume that all the s, c, and p are expressed as integer multiples of some basic indivisible time unit. Sporadic tasks make repeated requests for execution. Let t_k be the time that task T makes its k^{th} request for execution. The behavior of a sporadic task T is given by the following rules:

i) Task T makes its first request for execution at time $t_1 = s$.

ii) If T has period p, then T makes its $(k+1)^{st}$ request for execution at time $t_{k+1} \geq t_k + p \geq s + kp$. If task T makes its k^{th} execution request at time t_k, then the interval $[t_k, t_k+p]$ is called the k^{th} *request interval* (or simply a request interval).

iii) The k^{th} execution request of T must be completed no later than the *deadline* $t_k + p$.

iv) Each execution request of T requires c units of execution time.

We say that a task *misses a deadline* if an execution request of that task has not completed execution by its deadline. Sporadic tasks are independent in the sense that the time at which a task makes a request for execution is dependent only upon the time of that task's last execution request and not upon the the request times of any other task or tasks. A set of sporadic tasks τ is said to be *feasible* on a uniprocessor if it is possible to execute τ on a uniprocessor, subject to preemption constraints, such that every execution request of every task is guaranteed to have completed execution at or before its deadline. In guaranteeing response times to real-time tasks, we are only interested in determining feasibility, that is, determining if it is possible to guarantee a response time to each task that is less than or equal to its period. These guarantees will ensure that each execution request of each task will complete before its deadline.

We will assume that our real-time system contains m shared software resources $R_1, R_2, ..., R_m$. A software resource could be, for example, a pool of buffers, a portion of a database, or a global data structure. Whenever a task uses a shared resource, the task must be guaranteed exclusive access to the resource. Each execution request of task T_i consists of a sequence of n_i *phases* labeled $r_{i1}, r_{i2}, r_{i3}, ..., r_{in_i}$. The j^{th} phase of task i is represented by an integer r_{ij}, $0 \leq r_{ij} \leq m$, indicating the resource required by T_i during the j^{th} phase of its computation. We will assume that each phase of each task will require access to at most one resource. If $r_{ij} = 0$, then the j^{th} phase of T_i's computation requires no shared resources. Conceptually, if $r_{ij} = 0$, then the j^{th} phase of T_i requires the special resource R_0. Resource R_0 is the only resource that can be allocated to multiple tasks simultaneously. If a task never requires a resource (all phases use only resource R_0) then that task is called a *non-resource consuming task*. If a task ever requires a resource it is called a *resource consuming task*. We will assume that for each resource R_j, $1 \leq j \leq m$, there are at least two distinct tasks T_a and T_b such that $r_{ax} = r_{by} = j$. This is simply a requirement that each resource is in fact shared. The computational cost, c_i, of task T_i, is a given by:

$$c_i = \sum_{j=1}^{n_i} c_{ij} ,$$

where c_{ij} represents the computational cost of phase j. That is, c_{ij} is the time to execute phase j of task i to completion on a dedicated processor. If phase j requires a shared resource ($r_{ij} \neq 0$) then c_{ij} represents only the cost of using the resource and not the cost of accessing the resource. In later sections we will often wish to refer to the period of the "smallest" task that uses resource R_i. For resource R_i, let P_i represent this period. That is,

$$P_i = \underset{1 \leq j \leq n}{\mathrm{MIN}} (p_j \mid r_{jk} = i \text{ for some } k, 1 \leq k \leq n_j).$$

3. Programming Model

The concept of feasibility defined in the previous section is an *absolute* measure of temporal correctness. Feasibility is a property of a set of tasks which is independent of their implementation. However, it is often the case that in order to demonstrate that a set of tasks is feasible one must have a model of an implementation of a real-time system. In general, there are several factors to consider in the implementation of a real-time system. In a real-time system, as in most multiprogramming systems, there will be contention (queueing) for both the processor and for access to shared resources. Disciplines for controlling access to these resources need to be defined. We define an *implementation strategy* to be a scheme for synchronizing access to shared resources and for scheduling tasks. For a given implementation strategy, one can derive conditions under which a real-time system, implemented using this strategy, can be guaranteed to be viable. An implementation of a real-time system is *viable* if every execution request of every task can be guaranteed to complete execution at or before its deadline. Viability is a *relative* measure of temporal correctness. Viability is a property of tasks that is relative to a particular implementation strategy. To compare implementation strategies we need a notion of optimality. An implementation strategy is said to be *optimal* for a uniprocessor if every *feasible* real-time system implemented using this strategy can be guaranteed to be *viable*.

In this section we outline an implementation strategy based on monitors and deadline scheduling. In the sections that follow we will show that this implementation strategy is optimal for various characterizations of a tasks resource requirements. We first give a template for a task's structure.

We will assume that the body of a task T_x, is implemented according to the schema below. Conceptually, whenever a task makes an execution request, the following block of code is executed.

```
BEGIN
   ResourceRi.Request();     -- Phase 1
   < use resource $R_i$, $i = r_{x1}$ >
   ResourceRi.Release();

   ResourceRj.Request();     -- Phase 2
   < use resource $R_j$, $j = r_{x2}$ >
   ResourceRj.Release();
          :
          :
          :
   ResourceRk.Request();     -- Phase $n_x$
   < use resource $R_k$, $k = r_{xn}$ >
   ResourceRk.Release();
END
```

At the start of each phase, a task requests the resource it requires for that phase. At the end of each phase the task releases the resource it held during that phase. During a the lifetime of the system, a task is always in one of four states. Before a task is released it is *idle*. Each execution request of a task begins with an (implicit) request for the processor. Conceptually, this is the execution of the BEGIN statement above. While a task is waiting for the processor it is in the *ready* state. Once the processor is allocated to the task, then the task is in the *executing* state. If a task makes a request for a resource which is not available then the task will be *blocked.* At the end of the last phase the task conceptually executes the END statement. This returns the task to the idle state. The task will remain idle until the start of its next request interval. During the course of execution the task may be preempted. When a task is preempted it maintains possession of any resources it may have held. When preempted, a task returns to the *ready* state. We will assume that the BEGIN and END statements take no time to execute.[1]

Access to each resource R_i, $1 \leq i \leq m$, is controlled by a monitor that implements the request and release operations. A pseudo-code schema for such a monitor is shown below.

```
MONITOR ResourceRi =
   BEGIN
      -- Initialization
      var available : BOOLEAN := TRUE;
      var resource  : CONDITION;

   ENTRY PROCEDURE Request() =
      BEGIN
         WHILE( NOT available) DO
            WAIT( resource);
         END

         -- Caller has acquired exclusive
         -- access to Resource $R_i$.
         available := FALSE;
      END

   ENTRY PROCEDURE Release() =
      BEGIN
         available := TRUE;

         -- Wake-up all waiting tasks.
         BROADCAST( resource);
      END

   END
```

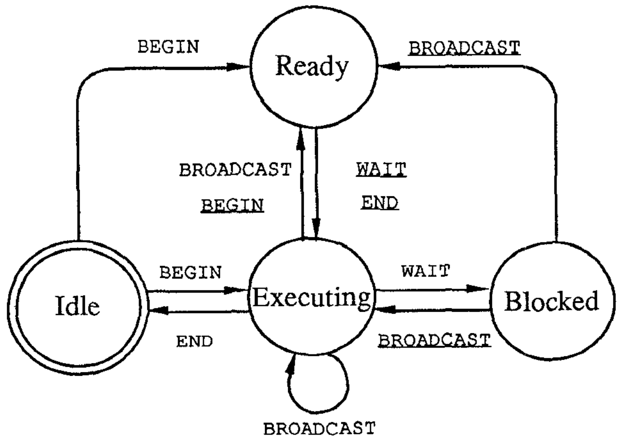
The semantics of the monitor ensures that only one task is ever executing code inside the monitor at any time. When a task wishes to gain access to resource R_i, it calls the Request entry of the monitor. Whenever a task returns from this call it is guaranteed to have exclusive access to the resource. If a task attempts to request a resource that is not available, then the task will execute the WAIT statement and be blocked on the resource's condition variable. When a task releases a resource it executes the BROADCAST statement which will wake-up all tasks (place them in the *ready* or *executing* state) that were blocked on the resource's condition variable. For resource R_0, assume that the request and release operations are null statements. We will assume that for all resources, the request and release operations take no time to execute.[2]

[1] The cost of the END statement could be included in the task's cost with little effort.

[2] In principle, the cost of these operations could be included in the cost of a task's phase. However, if these operations took time to execute then we would have to broaden the discussion to include details concerning the implementation of the WAIT and BROADCAST primitives. As such details are not central to the presentation of our results, we postulate a zero execution time cost.

An important feature of this synchronization scheme is that the broadcast operation does not explicitly allocate the newly freed resource to any task. Each task which is woken-up by a broadcast must re-attempt to acquire the resource on its own. (This style of synchronization is borrowed from the Mesa language [Lampson & Redell 80].) The decision as to which task acquires the resource will be under the control of the scheduler since it is the scheduler who decides which released task will execute first.

For our synchronization discipline, the following state transition diagram describes a task's behavior. The underlined operations are operations that are performed by other tasks. For example, the only way a task can leave the blocked state is for some other task to perform a broadcast. The BEGIN and END operations refer to the begin and end statements in the task schema above. Transitions to and from the idle and blocked states are initiated by the execution of these statements. A transition from the executing state to the ready state corresponds to a preemption of the executing task. Preemptions in the system occur only as the result of one of two actions. A task may be preempted by another task executing its BEGIN statement, or a task may be preempted when it performs a BROADCAST.



For our implementation strategy we propose a scheduling policy based on the *earliest deadline first* (EDF) selection rule. When making scheduling decisions, an EDF scheduler will dispatch the ready task whose deadline is nearest to the current value of real-time. Ties among tasks with the same deadline may be broken arbitrarily. In our implementation strategy, the scheduler makes scheduling decisions each time a task makes a request for execution (executes the BEGIN statement), requests an unavailable resource (performs a WAIT), completes a phase of its execution (performs a BROADCAST), or completes an execution request (executes the END statement).

Since our implementation strategy allows tasks to preempt one another, we need a slightly more sophisticated synchronization mechanism than the simple WAIT and BROADCAST operations described above. If preemption is allowed, then when a task T_i performs a request operation for resource R_k, it may be the case that R_k is allocated to a (preempted) task T_j whose current deadline is greater than that of task T_i. Since R_k is not available, T_i will become blocked on R_k's condition variable. To ensure that T_i's deadline can be respected, task T_j will have its deadline shortened to that of task T_i for the duration of T_j's current phase. This advancing of T_j's deadline occurs as a side effect of T_i's execution of the WAIT statement. In this manner, when T_i becomes blocked, T_j will have a deadline that is less than or equal to the deadlines of all the ready tasks. Therefore, under an EDF scheduler, T_j is quite likely to resume execution after T_i becomes blocked. When T_j performs the `release` operation at the end of its current phase, T_j will have its deadline restored (increased) to its original value as a side effect of the execution of the `BROADCAST` statement. At this point, all the tasks blocked on R_k's condition variable will become ready. The scheduler will preempt T_j and dispatch T_i (or some other task with a deadline less than or equal to T_i's deadline). This deadline advancement technique is a limited version of the scheme proposed in [Lampson & Redell 80] and [Sha et al. 87]. The analysis in the following sections will validate our implementation strategy.

4. Single Resource, Single Phase Systems

We begin with an analysis of the simplest real-time system. This is a system with a single shared resource R_1, and one where every task consists of only a single phase. This latter simplification means that if a task requires access to resource R_1, then each execution request of the task will require exclusive access to R_1 for the entire duration of its computation. For single phase systems, we will let r_i denote the resource requirement of the (single) phase of task T_i. In this section we derive necessary and sufficient conditions for such a real-time system to be viable on a single processor under our implementation strategy. We start by establishing necessary conditions for ensuring the viability of any implementation strategy whose scheduler does not use inserted idle time.[3]

Theorem 4.1: *Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a set of single phase sporadic tasks which share a single resource R_1. Assume that the tasks in τ are sorted in non-decreasing order by period. In the absence of inserted idle time, τ will be feasible for all possible release times only if:*

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

2) $\forall k,\ 1 \le k < n,\ r_k \ne 0$:
$$p_k \ge \underset{I_1}{\text{MAX}}\left(c_i + \underset{I_2}{\text{MAX}}\left(-l + \sum_{j=1}^{i-1}\left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j\right)\right),$$
where $I_1 = (k < i \le n) \wedge (r_i = r_k) \wedge (r_i \ne 0)$,
$I_2 = 0 < l < p_i - p_k$.

3) $\forall k,\ 1 \le k < n,\ r_k = 0$:
$$p_k \ge \underset{II_1}{\text{MAX}}\left(c_i + \underset{II_2}{\text{MAX}}\left(-l + \sum_{j=1}^{i-1}\left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j\right)\right),$$
where $II_1 = (k < i \le n) \wedge (r_i \ne 0)$,
$II_2 = \text{MAX}(0, P_1 - p_k) < l < p_i - p_k$.

Condition (1) is the requirement that the system not be overloaded (cumulative processor utilization less than 1 [Liu & Layland 73]). Conditions (2) and (3) have a less intuitive explanation. Condition (2) applies to resource consuming tasks and condition (3) applies to non-resource consuming tasks. If we think of a task's period as the maximum tolerable latency of each execution request, then informally, conditions (2) and (3) are requirements that each task's latency be greater than the worst delay that can be encountered while waiting to be scheduled. The intuition behind these conditions will be developed in the proof below (see also [Jeffay & Anderson 88]).

Note that a set of single phase sporadic tasks τ, where $r_i = 0$, for $1 \le i \le n$, corresponds to a set of tasks with no preemption constraints (independent tasks in the sense of [Liu & Layland 73]). In this case the above conditions reduce to condition (1) alone. This agrees with the result of [Liu & Layland 73]. A set of single phase sporadic tasks where $r_i = 1$, for $1 \le i \le n$, corresponds to a set of tasks which must be scheduled non-preemptively. In this case, the above conditions reduce to those reported in [Jeffay & Anderson 88].

Proof: We will actually prove a slightly stronger result, namely, that conditions (1), (2), and (3) are necessary to guarantee the feasibility of a set of *periodic* tasks. A periodic task is the special case of a sporadic task obtained when a sporadic task T_i makes execution requests every p_i time units.

Lemma 4.2: Let τ be a set of single phase periodic tasks which share a single resource R_1. Assume that the tasks in τ are sorted in non-decreasing order by period. In the absence of inserted idle time, τ will be feasible for all possible release times only if conditions (1), (2), and (3) hold.

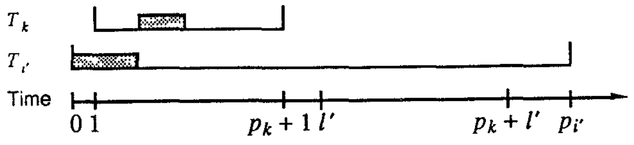
Proof: To show that conditions (1), (2), and (3) are necessary for all possible release times, we need only show that there exist one set of release times for which these conditions are necessary. We first show that condition (1) is necessary.

For all i, $1 \le i \le n$, let $s_i = 0$. Define $u_{a,b}$ as the total processor time consumed by τ in the interval $[a,b]$ when scheduled without inserted idle time. Let $t = p_1 \cdot p_2 \cdot \ldots \cdot p_n$. Consider the interval in time $[0,t]$. If τ is feasible then it must be the case that $u_{0,t} \le t$. Therefore, since $\frac{t}{p_i} c_i$ is the total processor time spent on task T_i in $[0,t]$, τ will be feasible only if

$$u_{0,t} = \sum_{i=1}^{n} \frac{t}{p_i} c_i \le t$$

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1.$$

An alternate explanation of conditions (2) and (3) is that for two tasks T_k and T_i, $k < i$, in all blocks of time of length $p_k + l$, $p_k < p_k + l < p_i$, there must exist enough unused processor time to execute T_i. To see that condition (2) is necessary, choose tasks T_k and $T_{i'}$, such that $k < i' \le n$, $r_k \ne 0$, $r_{i'} = r_k$, and a value l' such that $0 < l' < p_{i'} - p_k$. Let $s_{i'} = 0$, and $s_j = 1$, for $1 \le j \le n$, $j \ne i'$. This gives rise the pattern of task execution requests shown below.[4]



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

[4]In the scheduling diagrams used in this paper, a rectangle indicates an interval in which a task is executing. A rectangle open on the left side indicates that a task is being resumed. A rectangle open on the right side indicates that a task is being preempted. Tasks which require resources are represented with shaded rectangles. Tasks which do not require resources are represented with striped rectangles.

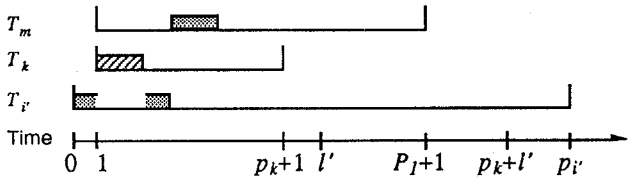
Because inserted idle time is not allowed, $T_{i'}$ will be dispatched at time zero. Since T_k and $T_{i'}$ both require the same resource (R_1), the execution request of $T_{i'}$ begun at time 0 must be completed by time $t = (p_k - c_k) + 1$ if τ is to be feasible. Therefore, in the interval $[0, l'+p_k]$, the total processor time consumed, $u_{0,l'+p_k}$, is bounded by

$$l'+p_k \geq u_{0,l'+p_k} \geq c_{i'} + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j ,$$

hence

$$p_k \geq c_{i'} - l' + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j .$$

To see that condition (3) is necessary, we use a construction similar to the one above. Choose tasks T_k and $T_{i'}$ such that $k < i' \leq n$, $r_k = 0$, $r_{i'} \neq 0$, and a value l' such that $\mathrm{MAX}(0, P_1 - p_k) < l' < p_{i'} - p_k$. Let $s_{i'} = 0$, and $s_j = 1$, for $1 \leq j \leq n, j \neq i'$. This gives rise the pattern of task execution requests shown below. Let T_m be the task with $r_m = 1$, whose period is P_1. (T_m is the task with the smallest period that accesses R_1.)



At time $t = 0$, task $T_{i'}$ may be preempted by task T_k (or by some other non-resource consuming task in with a nearer deadline). However, as in the previous case, $T_{i'}$ must still complete execution before time $t = (p_m - c_m) + 1$, if task T_m is to meet its deadline. Therefore, in the interval $[0, l'+p_k]$, the total processor time consumed, $u_{0,l'+p_k}$, is again bounded by

$$l'+p_k \geq u_{0,l'+p_k} \geq c_{i'} + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j ,$$

and hence

$$p_k \geq c_{i'} - l' + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j . \qquad \Delta$$

Returning to the proof of Theorem 4.1, since periodic behavior is a special case of sporadic behavior, any conditions necessary for the feasibility of a set of periodic tasks must also be necessary for the feasibility of a set of sporadic tasks. Therefore, by Lemma 4.2, in the of absence inserted idle time, conditions (1), (2), and (3) are necessary for the feasibility of sporadic tasks for all possible release times. This completes the proof of the theorem. Δ

Theorem 4.1 has given necessary conditions for the feasibility of a set of tasks for all possible release times. Often we are interested in determining the feasibility of a set of tasks for a specific set of release times. The following theorem can be used to show that in fact conditions (1), (2), and (3) of Theorem 4.1 are necessary feasibility conditions for sporadic tasks with arbitrary release times.

Theorem 4.3: A set of sporadic tasks τ, can be feasible for an arbitrary set of release times only if τ is feasible for all possible release times.

Proof: By the definition of sporadic tasks, a sporadic task will wait for an arbitrary amount of time between the end of one request interval and the start of the next. Therefore, after all tasks have been released, there can exist a time t such that a task, or group of tasks in τ, make requests for execution at time t, and there are no outstanding requests for execution at time t. In other words, if these tasks had not made execution requests at t then the processor would have been idle for some non-zero interval starting at t. At time t, τ is effectively "starting over" with a set of "release times" that are independent from the initial release times. Therefore, a set of sporadic tasks with arbitrary release times can be feasible only if they are feasible for all possible release times. Δ

The next theorem shows that conditions (1), (2), and (3) from Theorem 4.1, are sufficient for ensuring the viability of a set of single phase sporadic tasks that share a single resource under our implementation strategy. Since these conditions are necessary conditions for the feasibility of such a set of sporadic tasks, Theorem 4.4 also shows that the implementation strategy of Section 3 is an optimal strategy for resource consuming, single phase tasks which share a single resource. The optimality is with respect to the class of implementation strategies whose schedulers do not use inserted idle time. All of the optimality results in this paper will have this caveat. This optimality result means that if a set of single phase sporadic tasks which share a single resource can be viable under any implementation strategy which does not use inserted idle time in its scheduler, then the tasks must be viable under our strategy.

Theorem 4.4: Let τ be a set of single phase sporadic tasks as in Theorem 4.1. Under the implementation strategy of Section 3, τ will be viable for arbitrary release times, if conditions (1), (2), and (3) from Theorem 4.1 hold.

Proof: (By contradiction.) The proof proceeds by enumerating all possible types of blockage that a task can encounter. For each case we show that if a task misses a deadline then one the conditions from Theorem 4.1 must have been false. The full text of the proof appears in the appendix. Δ

5. Multiple Resource, Single Phase Systems

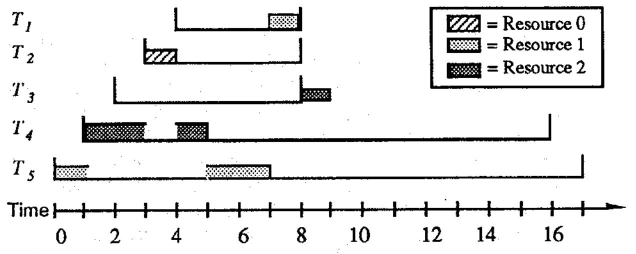
We next examine the problem of executing single phase sporadic tasks that share a set of resources. This problem differs from the previous problem in that it is now possible for resource consuming tasks to preempt one another provided they do not use the same resource. The main result of this section is to demonstrate that often it is inappropriate for an implementation strategy to allow these preemptions. Unfortunately, determining when it is appropriate to allow this additional preemption remains an open problem.

The implementation strategy in Section 3 uses a separate monitor to synchronize access to each resource. For a system with multiple resources, this does not lead to an optimal implementation strategy. Often a better strategy is to use a single monitor (with a single boolean and condition variable) to synchronize access to a group of resources. For our implementation strategy, an equivalent view of the problem is that it is often better to ignore the distinction between certain resources and to treat them as a single logical resource. Note that treating a group of resources as a single logical resource and using a single monitor to control access to these resources does not effect the logical correctness of the system. It can, however, effect the temporal correctness in interesting ways. The following example illustrates this phenomena.

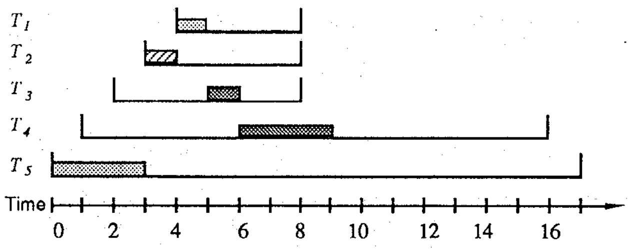
Example 5.1: Consider the following set of single phase tasks that share two resources R_1 and R_2.

$T_1 = (4,1,4)$, $r_1 = R_1$ $T_3 = (2,1,6)$, $r_3 = R_2$ $T_2 = (3,1,5)$, $r_2 = R_0$
$T_5 = (0,3,17)$, $r_5 = R_1$ $T_4 = (1,3,15)$, $r_4 = R_2$

Our implementation strategy will use a separate monitor for controlling access to each resource. This means that our strategy will allow preemptions between tasks that use different resources. Under our implementation strategy, the above tasks will be executed as shown below. Note that task T_3 misses a deadline at time $t = 8$.



Had we modified our characterization of these tasks so that R_1 and R_2 were treated as a single resource, then these same tasks could have been executed correctly by our implementation strategy. If we treat this multiple resource system as a single resource system by requiring all tasks requesting R_1 or R_2 actually request the meta-resource $R_{1\&2}$, then conditions (1), (2), and (3) from Theorem 4.1 hold for the resulting single resource system. By using a single monitor for controlling access to R_1 and R_2, we are disallowing preemption between the resource consuming tasks in this example. By disallowing this preemption, task T_4 would not have been scheduled at time 1 and T_3 will complete execution before its deadline as shown below.



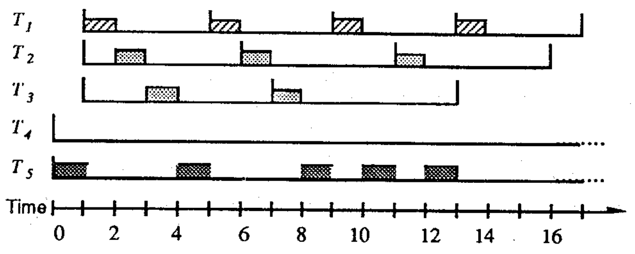
This anomalous behavior can be explained by closely examining the effect of preemption among resource consuming tasks. In the previous section, the worst case delay that a task T_k could experience while waiting to be scheduled, occurred when a *single* resource consuming task T_i with a larger period made an execution request l time units before T_k's request. Under the right circumstances, T_k had to wait for the execution request of this larger task to complete before T_k was allowed to complete. By allowing resource consuming tasks to preempt one another, it is now possible for T_k to have to wait for ***multiple*** resource consuming tasks with further deadlines to complete before being scheduled. For the specific tasks above, we do better by prohibiting this behavior from occurring.

However, always disallowing preemption among resource consuming tasks does not lead to an optimal implementation strategy. Under certain circumstances, allowing resource consuming tasks to preempt one another makes previously unviable tasks sets viable.

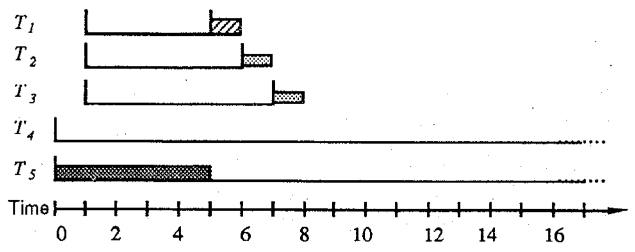
Example 5.2: Consider the following set of single phase tasks that share two resources R_1 and R_2. For the implementation strategy of Section 3 (separate monitors are used for accessing R_1 and R_2), the following tasks can be shown to be viable. (The analysis to show that these tasks will be viable will be presented below. The figure below is merely intended to suggest viability.)

$T_1 = (1,1,4)$, $r_1 = R_0$ $T_2 = (1,1,5)$, $r_2 = R_1$ $T_4 = (0,5,25)$, $r_5 = R_2$
$T_3 = (1,1,6)$, $r_3 = R_1$ $T_5 = (0,5,28)$, $r_4 = R_2$

Under our implementation strategy, these tasks will be executed as shown below.



However, had we ignored the distinction between resources R_1 and R_2 and used a single monitor for accessing these resources, then these tasks would have been quite unviable under our implementation strategy.



An important difference between the tasks in these two examples is that in the second example, the periods of the tasks that used R_1 were all less than the periods of the tasks that used resource R_2. Because of this, for any request of task T_k, $r_k \neq 0$, it is not possible for more than one resource consuming task with a further deadline to execute while T_k has an outstanding request for execution. In order for multiple resource consuming tasks with further deadlines to execute while T_k has an outstanding request for execution, there must exist tasks T_i and T_j with $r_i = r_k$, $r_j \neq r_k$ $(\neq 0)$ and $p_k < p_j < p_i$.

A precise characterization of feasibility and viability conditions for single phase tasks that share a set of resources, is the object of an on-going study. One special case that has been solved is the case where there is no overlap in the periods of the tasks that consume resources. That is, if tasks T_k and T_i, $k < i$, require resource R_a, $a \neq 0$, and there does not exist a task T_j that requires resource R_b, $b \neq a \neq 0$, such that $p_k < p_j < p_i$, then we can determine the task's feasibility.

Theorem 5.1: Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a set of single phase sporadic tasks which share m resources R_1 - R_m. Assume that the tasks in τ can be labeled such that if $k < i$, then $p_k \le p_i$, and if $k < j < i$, and $r_k = r_i$ ($\neq 0$), then $r_k = r_j = r_i$. In the absence of inserted idle time, τ will be feasible only if:

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

2) $\forall k,\ 1 \le k < n,\ r_k \neq 0$:

$$p_k \ge \underset{I_1}{\mathrm{MAX}}\left(c_i + \underset{I_2}{\mathrm{MAX}}\left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right)\right),$$

where $I_1 = (k < i \le n) \wedge (r_i = r_k) \wedge (r_i \neq 0)$,
$I_2 = 0 < l < p_i - p_k$.

3) $\forall k,\ 1 \le k < n$:

$$p_k \ge \underset{II_1}{\mathrm{MAX}}\left(c_i + \underset{II_2}{\mathrm{MAX}}\left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right)\right),$$

where $II_1 = (k < i \le n) \wedge (r_i \neq r_k) \wedge (r_i \neq 0)$,
$II_2 = \mathrm{MAX}(0, P_{r_i} - p_k) < l < p_i - p_k$.

Proof: The necessity of these conditions follow immediately from the related constructions in the proof of Theorem 4.1. Δ

Note that condition (3) now applies to both resource consuming and non-resource consuming tasks. For multiple resource systems, it can be shown that the implementation strategy of Section 3 is an optimal strategy for single phase sporadic tasks with no overlap in the periods of the tasks that consume resources. As in the previous section, the optimality is shown by demonstrating that the conditions necessary for feasibility are sufficient conditions for ensuring the viability of our implementation strategy.

Theorem 5.2: Let τ be a set of single phase sporadic tasks as in Theorem 5.1. Under the implementation strategy of Section 3, τ will be viable for arbitrary release times, if conditions (1), (2), and (3) from Theorem 5.1 hold.

Proof: The proof is largely identical to the proof of Theorem 4.4 and will not be repeated here. Δ

For sets of single phase tasks with arbitrary overlap in the periods of resource consuming tasks, the above results suggest a heuristic for ensuring viability. A set of meta-resources is created by treating groups of resources as a single resource in such a manner that there no longer is any overlap in the periods of meta-resource consuming tasks. The conditions from Theorem 5.1 can then be used to determine viability of the transformed system.

6. Multiple Phase Systems

We finally consider the problem of determining feasibility conditions for a set of sporadic tasks when each task consists of a set of phases. This characterization addresses tasks which require multiple resources to execute. (Recall, however, that we are still assuming that each phase of a task requires at most one resource.) The main result of this section is to show that the analysis of the previous sections is sufficient to analyze these systems. We will show how a multiple phase task can be thought of as a set of single phase tasks and how our implementation strategy for single phase tasks can be modified to execute multiple phase tasks.

Let τ be a set of multiple phase sporadic tasks. We can create an equivalent set τ', of single phase sporadic tasks such that τ' will be feasible if and only if τ is feasible. The set τ' is constructed as follows. For a task T_k in τ with n_k phases, we will create n_k single phase tasks $T_{ki} = (s_k, c_{ki}, p_k)$, $1 \le i \le n_k$. All tasks derived from T_k will make requests for execution at the same points in time at which T_k would have made execution requests. These tasks can be scheduled as ordinary single phase tasks except that for each T_k in τ, a precedence order must be maintained between tasks T_{ki}, $1 \le i \le n_k$, in τ'. Each execution request of task T_{ki}, $i > 1$, made at time t, cannot be scheduled, or allocated resources, until after the execution request of task T_{ki-1} made at time t has completed execution.

To execute the set of single phase sporadic tasks τ', we supplant the scheduler in our implementation strategy with similar scheduler based on a refined EDF scheduling policy that we will call the EDF* policy. An EDF* scheduler behaves exactly as an EDF scheduler except when choosing among tasks with the same deadline. Recall that the EDF rule allows for an arbitrary choice among tasks with the same deadline. We will exploit this feature to enforce the precedence constraints on the tasks in τ'. When there are multiple ready tasks with the nearest deadline, the EDF* policy will choose a ready task T_{ki} only if there does not exist another task T_{kj}, $j < i$, in the ready or blocked state. For our modified implementation strategy, the transformation from a set of multiple phase sporadic tasks, to a larger set of single phase sporadic tasks, outlined above is correct in the sense that for each k, the aggregate behavior of the tasks T_{ki}, $1 \le i \le n_k$, will be indistinguishable from the behavior of T_k. Such an implementation strategy can correctly execute τ' if and only if it can correctly execute τ.

Unfortunately, the problem of deciding feasibility for τ' is more complicated than the problem of deciding feasibility for single phase tasks. Recall that we have assumed that the times at which tasks made execution requests were independent. The times at which the single phase tasks derived from a multiple phase task, make execution requests are not independent. For all k, tasks T_{k1} - T_{kn_k} will always make execution requests at the same time. This fact must be reflected in the feasibility analysis since these tasks can *never* interfere with each other. They will never compete for resources or block one another.

Therefore, the feasibility conditions from the previous sections are only sufficient conditions for the feasibility of a single phase system derived from a multiple phase system. They must be generalized slightly in order to become necessary conditions.

The following theorem extends Theorem 4.1 for multiple phase tasks. It establishes necessary conditions for the feasibility of multiple phase tasks that share a single resource. This corresponds to a set of tasks where in each request for execution, a task alternates between using the resource and not using the resource.

Theorem 6.1: Let $\tau = \{T_{11}, T_{12}, \ldots, T_{1n_1}, T_{21}, T_{22}, \ldots, T_{2n_2}, \ldots, T_{n1}, T_{n2}, \ldots, T_{nn_n}\}$ be a set of single phase sporadic tasks derived from a set of multiple phase sporadic tasks $\{T_1, T_2, \ldots, T_n\}$ that share a single resource R_1. Assume τ is sorted such that if $i < j$, then $p_{ix} \le p_{jy}$ for all x and y, $1 \le x \le n_i$ and $1 \le y \le n_j$ (recall that for all k, $1 \le k < n_i$, $p_{ik} = p_i$). For tasks T_{i1} - T_{in_i}, define the cost function $C_i(h)$ to be

$$C_i(h) = \begin{cases} 0 & \text{if } h = 1, \\ \sum_{j=1}^{h-1} c_{ij} & \text{if } 1 < h \le n_i\ . \end{cases}$$

In the absence of inserted idle time, τ will be feasible for arbitrary release times only if:

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

2) $\forall k,\ 1 \le k < n,\ \forall g,\ 1 \le g < n_k$: $r_{kg} \ne 0$:

$$p_k \ge \underset{i,h:\,I_1}{\mathrm{MAX}} \left(c_{ih} + \underset{l:\,I_2}{\mathrm{MAX}} \left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right) \right),$$

where $I_1 = (k < i \le n) \wedge (1 \le h \le n_i) \wedge (r_{ih} \ne 0) \wedge (r_{ih} = r_{kg})$,
$I_2 = 0 < l < (p_i - p_k) - C_i(h)$,

3) $\forall k,\ 1 \le k < n,\ \forall g,\ 1 \le g < n_k$: $r_{kg} = 0$:

$$p_k \ge \underset{i,h:\,II_1}{\mathrm{MAX}} \left(c_{ih} + \underset{l:\,II_2}{\mathrm{MAX}} \left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right) \right),$$

where $II_1 = (k < i \le n) \wedge (1 \le h \le n_i) \wedge (r_{ih} \ne 0)$,
$II_2 = \mathrm{MAX}(0, P_1 - p_k) < l < (p_i - p_k) - C_i(h)$.

Conditions (2) and (3) are semantically equivalent to the corresponding conditions in Theorem 4.1. These conditions differ from those of Theorem 4.1 in that for a task T_{kg}, conditions (2) and (3) do not include any delay due to *waiting* for other tasks T_{kj} to complete execution. As in Theorem 4.1, condition (3) only applies to those tasks that never use any resources.

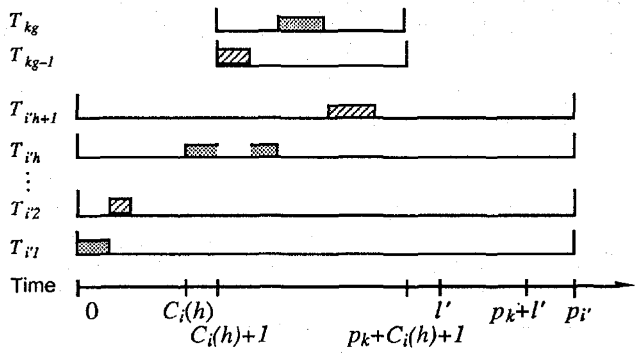
Conditions (2) and (3) also differ in that the range of the lag time parameter l is more restricted than in Theorem 4.1. This is a reflection of the existence of a precedence relation among subsets of the tasks. For the previous problems, the worst case blockage of a task T_k occurred when a task T_i with a larger period commenced execution l time units before a request interval of T_k. In those cases, the maximum that the lag could be was $p_i - p_k - 1$ time units. In the current problem, since there exists a precedence relation on tasks, task T_{ih} can never be scheduled until $C_i(h)$ time units after it makes an execution request. The cost function $C_i(h)$ represents the cost of the execution of the $h - 1$ tasks that must precede each execution request of task T_{ih}. Therefore, when assessing the blockage due to a task T_{ih} executing just prior to an execution request of a task T_{kg} with a smaller period, we need only consider a maximum lag between these two tasks of

$$l < (p_i - p_k) - C_i(h)$$

time units.

Proof: We will only demonstrate the necessity of conditions (2) and (3).

For condition (2), choose tasks T_{kg} and $T_{i'h}$, such that $k < i' \le n$, and $1 \le h \le n_i$, $1 \le g \le n_k$, $r_{ih} \ne 0$, and $r_{ih} = r_{kg}$. $T_{i'h}$ is a task that requires the same resource that T_{kg} needs. Choose a value l' such that $0 < l' < (p_{i'} - p_k) - C_i(h)$. Let $s_{i'} = 0$, and $s_j = C_i(h) + 1$, for $1 \le j \le n, j \ne i'$. This gives rise the pattern of task execution requests shown below.[5] (Recall that shaded cost rectangles denote resource phases.)



Following the analysis of Lemma 4.2, we have in the interval $[0, l'+p_k]$, the total processor demand $d_{0,l'+p_k}$, bounded by

$$l' + p_k \ge d_{0,l'+p_k} \ge c_{i'h} + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k + l' - 1}{p_j} \right\rfloor c_j,$$

hence

$$p_k \ge c_{i'h} - l' + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k + l' - 1}{p_j} \right\rfloor c_j .$$

The construction showing the necessity of condition (3) is similar. Δ

The next theorem shows that the above conditions are sufficient for the ensuring the viability of a set of tasks under our implementation strategy. Therefore, the implementation strategy in Section 3, modified with an EDF* scheduler, is an optimal strategy for multiple phase tasks that share a single resource.

Theorem 6.2: Let τ be a set of multiple phase sporadic tasks as in Theorem 6.1. Under the implementation strategy of Section 3 with an EDF* scheduler, τ will be viable for arbitrary release times, if conditions (1), (2), and (3) from Theorem 6.1 hold.

Proof: Excluding some minor changes in notation, the proof is identical to the proof of Theorem 4.4 and will not be repeated here. Δ

Although we have only considered multiple phase tasks that share a single resource, we can draw several conclusions that will be true for all multiple phase systems. First, note that the feasibility of a set of multiple phase tasks is not a function of the number of phases of the tasks. What is important is the relative sizes of only the resource consuming phases. The relative sizes of the non-resource consuming phases is immaterial. For example, adjacent non-resource consuming phases may be coalesced without affecting the feasibility of the system. Of course resource consuming phases should be as short as possible. Therefore, there is a potential benefit, in terms of feasibility, to separating a resource consuming phase into multiple phases. This may be possible if, for example, a phase consists of a sequence of disjoint operations. This technique amounts to defining allowable preemption points within a resource consuming phase.

[5] Phases within a task are demoted by separate cost rectangles.

7. Discussion and Related Work

The request and release operations in our implementation strategy can be thought of as an implementation of the P and V operations, respectively, on a binary semaphore. The boolean variable in the monitor functions as the binary semaphore. There are two important features of our implementation of these operations which were critical to the optimality of our strategy. The first feature is that resources are never assigned to a task by another task. Since each task must explicitly acquire a resource on its own, a task must be executing when it acquires a resource. Therefore, in our idealized environment, a task is guaranteed to execute for at least one time unit immediately after acquiring the resource. Put another way, resources are allocated to tasks as late as possible.

The second notable feature of our implementation is that available resources are always acquired by the ready task with the nearest deadline (of all the ready tasks in the system). The feature is due to our use of the `BROADCAST` primitive. Note that it would be quite awkward to guarantee that an implementation based on the more common `WAIT` and `SIGNAL` synchronization primitives would have the features above. This is because the signal operation only releases a single waiting task [Hoare 74]. For example, if the resource is allocated to the signaled task, then problems arise if a task with a nearer deadline, which desires the same resource, becomes ready before the signaled task resumes execution.

Our implementation strategy has the additional interesting property that a task will execute the wait statement in the request operation at most once for any resource request. This is due to the fact that we use deadline scheduling. Once a task T_k has been blocked and is made ready by a `BROADCAST`, T_k will execute when it has the nearest deadline. In the meantime the resource it requires will be used only by tasks with a nearer deadline (which will execute before T_k). When T_k is dispatched the resource it requested must be available. Therefore, the loop in the `request` monitor entry can be replaced by an *if* statement. This observation is important for assessing the cost (overhead) of our synchronization scheme. Although we are currently ignoring the cost of the `request` and `release` operations, this observation simplifies the determination of their cost since the loop is executed at most once.

Another issue related to the overhead of our implementation strategy concerns the cost of preemption. In our strategy, tasks which require different resources may preempt one another. While we have been ignoring the cost of this preemption, in practice there is a high cost associated with a context switch. Therefore, it would be useful to determine, for a given real-time system, if the preemption in our implementation strategy is necessary for viability. The analysis of the previous sections can be used to determine the minimum amount of preemption necessary for ensuring the viability of a set of tasks. For example, any set of tasks can be logically implemented using a single monitor for accessing all resources (including R_0). This results in a non-preemptive implementation. Such an implementation is desirable to use whenever possible as it is simple and efficient (in terms of overhead). The conditions of Theorem 4.1 are necessary and sufficient for the viability of this strategy. Task sets which are not viable under this strategy can be implemented using a monitor for resource consuming tasks and a monitor for non-resource consuming tasks, or a monitor for a group of resource consuming tasks with no overlap among the periods of the tasks. For each of these characterizations of a real-time systems resource requirements, the viability conditions (necessarily) become less restrictive as more preemption is allowed.

The optimality of our results are quite dependent on our characterization of a task's behavior. When tasks make requests for execution at constant intervals (tasks are *periodic*), Mok has shown that the problem of deciding whether or not it is possible to execute a set of cyclic tasks which use semaphores to enforce mutual exclusion is NP-hard in the strong sense [Mok 83]. In [Jeffay and Anderson 88] the more general problem of deciding whether or not it is possible to execute a set of periodic tasks in a non-preemptive manner was also shown to be NP-hard in the strong sense. In addition, it was shown that if an optimal non-preemptive scheduling algorithm existed for periodic tasks, then P = NP [Jeffay & Anderson 88]. For periodic tasks, the intractability arises from our inability to efficiently determine if the worst case blockage that a task may experience while waiting to execute, can ever actually occur. The optimality of the results in this paper is primarily due to the fact that we are allowing a small amount of non-determinism in the behavior of tasks. Since sporadic tasks may delay for an arbitrary interval between making execution requests, we can argue that a sporadic task can always experience its worst case blockage. Since sporadic tasks are a generalization of periodic tasks, all of the feasibility conditions we developed for sporadic tasks will be sufficient conditions for the viability of periodic tasks under our implementation strategy.

Given the intractability results for periodic tasks, it is not surprising that previous work in this area has focused on heuristic solutions. One approach 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 independent periodic tasks [Mok et al. 87, Sha et al. 87]. In this manner, the results developed for independent periodic tasks can be applied. For independent periodic tasks, the conditions which are necessary and sufficient for guaranteeing response times are stated in terms of the processor utilization of the system. Independent tasks can be viable if

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

where the value of α 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 viability conditions for task sets with preemption constraints is $U \leq \alpha - B$, where B is a function of the durations that tasks in the system can be blocked [Mok et al. 87, Sha et al. 87]. The reduction process results in conditions which are sufficient for ensuring the temporal correctness of a set of tasks but which are not necessary. In effect, these methods are sacrificing processor utilization to gain viability.

Our approach is based on an examination of the relative sizes of the task's periods and costs and not on utilization. The majority of our analysis has been directed at determining if a task's period is large enough to accommodate the blockage due to other tasks' executions. The constraints we impose on a task's period (e.g., conditions (2) and (3) in Theorem 4.1) are not a function of processor utilization. It is also the case that processor utilization is not directly effected by these constraints. For example, for the feasibility conditions of Theorem 4.1, it is possible to have task sets which satisfy condition (1) (a utilization constraint) but which do not satisfy condition (2) or (3). Similarly, it is possible to have task sets which satisfy conditions (2) and (3), but which do not satisfy condition (1). We conclude from these observations that the viability conditions of task sets with preemption constraints need not be considered a function of processor utilization. One does not have to trade-off utilization to ensure viability.

Other work in this area has focused on the analysis of specific implementation strategies [Leinbaugh 80, Stoyenko 87]. These researchers have also developed conditions which are sufficient for ensuring the correctness of cyclic tasks which share resources. Our work extends theirs by considering a more general characterization of a real-time task, and by deriving necessary conditions for the feasibility of various resource usage patterns. A more pragmatic approach to guaranteeing response times to tasks has appeared in [Zhao et al. 87a, 87b]. They primarily consider tasks which make a single request for execution and have resource requirements and deadlines. They also consider a distributed processing environment. Scheduling and load balancing heuristics are presented and analyzed.

8. Summary and Conclusions

In this paper we have considered the problem of guaranteeing response times to sporadic tasks with non-preemptable resource requirements. We sought to guarantee that each task finish processing before its deadline. When posing solutions to this problem, details concerning the implementation of the system must be considered. We have developed a model of an implementation that we termed an *implementation strategy*. An implementation strategy consists of a scheme for synchronizing access to shared resources and for scheduling tasks. We have presented and analyzed an implementation strategy based on monitors, with wait and broadcast primitives, and deadline scheduling. We have shown that this strategy is an optimal strategy for single and multiple phase sporadic tasks that share a single resource. The optimality is with respect to the class of implementation strategies whose schedulers do not use inserted idle time. For tasks that share more than one resource, we have shown that often allowing preemption between resource consuming tasks can lead to sub-optimal strategies. If there is no overlap in the periods of the resource consuming tasks, then our implementation strategy is again an optimal strategy. If there is an overlap in the periods, then the determination of necessary conditions for the feasibility of multiple resource single phase systems remains an open problem. Our viability conditions are sufficient for these systems. Lastly, we have shown how multiple phase systems can be modeled by a single phase system with precedence constraints. Our implementation strategy can be easily extended to incorporate these precedence constraints. While we have focused on a model of sporadic tasks, all of our results are applicable (as sufficient conditions) to periodic tasks.

This work makes two contributions. First, we have shown that response times can be guaranteed to tasks with preemption constraints without having to trade-off processor utilization to achieve viability. The analysis presented is based on an examination of the relationships of task's periods and costs. It shows that in principle, feasibility is not a function of processor utilization. One can conceive of task sets with arbitrarily high processor utilization (although less than or equal to 1.0 of course), which will be viable.

The second contribution of this work is the development of a family of implementation strategies for sporadic tasks with preemption constraints. When considering how best to execute a set of tasks it is useful to know how much preemption must be implemented in order to guarantee viability. This is an important measure to quantify because preemption often has a high cost associated with it. We have analyzed implementations strategies which allow no preemption, strategies which allow preemption between tasks that use resources and those that do not, and strategies which allow preemption between resource consuming tasks.

Future work in this area is targeted at tasks with nested resource requirements. It is interesting to note that this paper already contains two simple but useful solutions to this problem. If we assume that a task's resource requests within a phase are ordered (to avoid deadlock), then we can guarantee feasibility by either disallowing all preemption or by ignoring the distinction between resource types. The run-time systems of Sections 4 and 6 are sufficient for handling these cases.

9. References

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

[Jeffay & Anderson 88]
Jeffay, K., Anderson, R., *On Optimal, Non-Preemptive Scheduling of Periodic and Sporadic Tasks,* University of Washington, Department of Computer Science, Technical Report #88-11-06, November 1988. (Submitted for publication.)

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

[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.

[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.

[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.

[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. IEEE Real-Time Systems Symp., San Jose, CA, December 1987, pp. 133 - 143.

[Sha et al. 87] Sha, L., Rajkumar, R., Lehoczky, J.P., *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, Carnegie Mellon University, Department of Computer Science, Technical Report CMU-CS-87-181, November 1987.

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

[Zhao et al. 87a]
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.

[Zhao et al. 87b]
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.