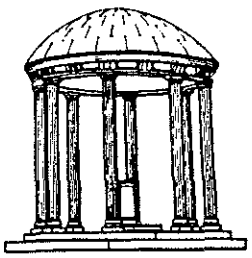
On Optimal, Non-Preemptive Scheduling of Periodic and Sporadic Tasks

TR90-019

April, 1990

Kevin Jeffay
Richard Anderson
Charles U. Martel



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

On Optimal, Non-Preemptive Scheduling of Periodic and Sporadic Tasks

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

Richard Anderson**
University of Washington
Department of Computer Science and Engineering, FR-35
Seattle, WA 98195

Charles U. Martel***
University of California at Davis
Computer Science Division
Davis, CA 95616

Technical Report 90-019
March 1990

Abstract: Periodic and sporadic tasks are central components in both the analysis and implementation of real-time systems. A periodic task makes requests for execution at precise intervals while a sporadic task makes execution requests at arbitrary times but with a bounded minimum duration between requests. This paper examines the problem of scheduling a set of periodic or sporadic tasks on a uniprocessor using a non-preemptive discipline. We derive a set of conditions to guarantee the correctness of a non-preemptive deadline driven scheduling algorithm. We then show that for scheduling sporadic tasks this discipline is optimal over the class of non-preemptive algorithms which do not use inserted idle time. The problem of determining feasibility for our algorithm can be decided in pseudo-polynomial time. This scheduling discipline is also optimal for scheduling periodic tasks when all possible release times are considered. Lastly, we show that if there exists an optimal polynomial time scheduling algorithm for periodic tasks with arbitrary release times, then P = NP.

---

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

** Supported in part by a National Science Foundation Presidential Young Investigator Award and in part by the Digital Equipment Corporation External Research Program.

*** Supported in part by a grant from the National Science Foundation (number CCR-8722848).

1. Introduction

The concept of a task that repeatedly makes requests for execution is central to both the design and analysis of real-time systems. In particular, formal studies of real-time systems frequently represent the time constrained processing requirements of the system as a set of *periodic* or *sporadic* tasks with deadlines [Liu & Layland 73, Leung & Merrill 80, Mok 83, Jeffay 89a]. A periodic task will make execution requests at regular intervals while a sporadic task makes execution requests at arbitrary times but with a bounded minimum duration between requests. In practice, periodic tasks are commonly found in application such as avionics and process control, in which accurate control requires sampling and processing data at precise intervals. Sporadic tasks are also used in these applications but are more commonly associated with event driven processing such as processing user inputs or non-periodic device interrupts.

Given a real-time system, the goal is to determine whether or not it is possible to schedule the system's tasks on a processor, or processors, such that each task completes execution before some well defined deadline. In this paper we consider the problem of non-preemptively scheduling a set of periodic or sporadic tasks on a uniprocessor. The emphasis on non-preemptive scheduling is motivated by the following observations. In many practical real-time scheduling problems, preemption is not allowed. For example, in I/O scheduling, properties of device hardware and software often either prohibit preemption or have a prohibitive cost associated with preemption. Secondly, most formal models of a real-time system assume tasks are independent and do not share resources. Actual systems rarely meet these assumptions. Non-preemptive scheduling can provide a simple but effective vehicle for ensuring mutually exclusive access to shared resources and data in a uniprocessor system. The problem of scheduling all tasks non-preemptively forms the basis for more general tasking models that include shared resources [Jeffay 89b]. Finally, in practice, a non-preemptive scheduling discipline is more desirable than a preemptive discipline since a non-preemptive discipline is easier to implement and can exhibit dramatically lower overhead at run-time. More importantly, it is easier to characterize the overhead of dispatching tasks in the non-preemptive case. Since scheduling overhead is typically ignored in most scheduling models (including ours), an implementation of a non-preemptive scheduler will be closer to the formal model than an implementation of a preemptive scheduler.

Formally, a periodic or sporadic task T is a 3-tuple (s, c, p) where

s = release time: the time of the first request for execution of task T,

c = computational cost: the amount of processor time required in the worst case to execute task T to completion on a dedicated uniprocessor, and

p = period: the interval between requests for execution of task T.

Throughout this paper we assume a discrete time model. In this domain we assume that all s, c, and p are expressed as integer multiples of some indivisible time unit.

The behavior of a *periodic* task T is given by the following execution rules. Let t_k be the time that task T makes its k^{th} request for execution.

i) If task T has period p and makes its first request for execution at time s, then for all $k \geq 1$, T will make its k^{th} request for execution *exactly* at time $t_k = s + (k-1)p$.

ii) The k^{th} execution request of task T must be completed no later than the *deadline* $t_k + p = (s + (k-1)p) + p = s + kp = t_{k+1}$.

iii) Each execution request of task T requires a constant c units of execution time.

If an execution request of a task has a deadline at time t_d, and the request has not completed execution at time t_d, then we say the task has missed a deadline. If task T makes its k^{th} execution request at time t_k, then the closed interval $[t_k, t_k+p]$ is called the k^{th} *execution request interval* (or simply a *request interval*) of task T.

A sporadic task is a generalization of a periodic task. The behavior of a sporadic task T is slightly less constrained than a periodic task. Its behavior is given by the following rules.

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

ii) If task T has period p, then for all $k \geq 1$, T makes its $(k+1)^{st}$ request for execution at time $t_{k+1} \geq t_k + p \geq s + kp$.

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 a constant c units of execution time.

The "period" of a sporadic task is simply the minimum time between any two successive execution requests of the task. We assume sporadic tasks are independent in the sense that the time of a task's execution request is dependent only upon the time of its last request and

not upon those of any other task. Once released, both periodic and sporadic tasks make requests for execution forever.

A set of periodic or sporadic tasks τ, is said to be *feasible* on a uniprocessor if it is possible to schedule τ on a uniprocessor, without preemption, such that every execution request of every task T is guaranteed to complete execution at or before its deadline. A non-preemptive scheduling discipline is said to be *optimal* for a uniprocessor if it can correctly schedule any task set that is feasible on a uniprocessor.

We will show that for sporadic tasks, a deadline driven scheduling algorithm that is a non-preemptive version of the Earliest Deadline First (EDF) [Liu & Layland 73], is optimal with respect to the class of algorithms that do not use inserted idle time.[1] For periodic tasks, our ability to decide feasibility will be a function of our knowledge of the tasks' release times. When all possible release times are considered, the non-preemptive EDF discipline will also be shown to be optimal for periodic tasks. However, for a given set of release times, we show that if there exists an optimal non-preemptive scheduling discipline for periodic tasks with arbitrary release times that takes a only polynomial amount of time to make each scheduling decision, then P = NP. This suggests that there likely does not exist an optimal non-preemptive discipline for scheduling periodic tasks with arbitrary release times.

Previous work in the area of real-time scheduling has mainly focused on the analysis of preemptive scheduling disciplines. For the case where all tasks are periodic and have release times of 0, a well known result is that the preemptive EDF algorithm is optimal [Liu & Layland 73]. Allowing the release times to be arbitrary integers does not change the result [Jeffay 89a]. The extension of the preemptive problem to multiprocessors was considered in [Dhall & Liu 78] and [Bertossi & Bonuccelli 83]. Work with non-preemptive disciplines has typically been confined to the consideration of models where tasks make only single requests for execution and there is a precedence order between the tasks. In addition, each task's execution request requires only a single unit of computation time and must be completed before a deadline [Garey et. al 81, Frederickson 83].

---

[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].

A more general characterization of periodic tasks has been considered in [Leung & Merrill 80], [Lawler & Martel 81], [Leung & Whitehead 82], and [Mok 83]. In these works, when a task makes an execution request, it may have a deadline nearer than the time of the next execution request. For this more general model, Mok has shown that the problem of deciding feasibility of a set of periodic tasks which use semaphores to enforce mutual exclusion constraints is NP-hard [Mok 83]. This paper demonstrates the intractability of the feasibility question for an even simpler characterization of periodic tasks and provides strong evidence that there may not exist an optimal non-preemptive scheduling discipline for periodic tasks.

The remainder of this paper is composed of three major sections. In the following section we prove the optimality of the non-preemptive EDF algorithm, over all algorithms which do not use inserted idle time, for sporadic tasks and for periodic tasks when all possible release times are considered. We are also interested in the complexity of deciding if a set of tasks is feasible. Section two also presents a pseudo-polynomial time algorithm for deciding if a set of sporadic tasks is feasible when scheduled without inserted idle time. Section three demonstrates the absence of an optimal algorithm for periodic tasks with arbitrary release times and proves that the problem of deciding feasibility of a periodic task set (with arbitrary release times) is intractable. Section four discusses these results.

2. Optimal Non-Preemptive Scheduling of Sporadic Tasks

The basic scheduling policy we consider is the *earliest deadline first* (EDF) policy [Liu & Layland 73]. The EDF scheduling policy dictates that when selecting a task for execution, the task with an uncompleted execution request whose deadline is closest to the current point in time is chosen for execution. Ties between tasks with identical deadlines are broken arbitrarily. Once scheduled, a task is immediately executed to completion without preemption. Unless the processor is idle, a non-preemptive scheduler will make dispatching decisions only when a task terminates an execution request. If the processor is idle then the first task to make an execution request is scheduled. We assume that both the task selection process and the process of dispatching a task take no time in our discrete time system.

In this section we derive conditions that ensure the correctness of the non-preemptive EDF algorithm for scheduling periodic and sporadic tasks on a uniprocessor. We also show the

optimality of the non-preemptive EDF algorithm for sporadic tasks. The non-preemptive EDF algorithm is also shown to be optimal for periodic tasks when all possible release times are considered. Throughout this section, all optimality claims are assumed to be claims of optimality with respect to the class of non-preemptive scheduling algorithms that do not use inserted idle time. At the end of this section we will briefly comment on the problem of scheduling with inserted idle time.

2.1 Scheduling Without Inserted Idle Time

In [Jeffay 89a] it was shown that when preemption is allowed at arbitrary points, the feasibility of a set of periodic or sporadic tasks was independent of the tasks' release times. When preemption is not allowed, the release time of a task plays a more significant role in the analysis of non-preemptive feasibility. We start by deriving feasibility conditions for all possible release times and then extend these results to cover arbitrary release times. These two problem characterizations correspond to problem domains in which the release times of tasks are unknown and known, respectively. We first develop necessary conditions for feasibility and then show these conditions are sufficient for ensuring the correctness of the non-preemptive EDF discipline. Our approach is motivated by the early work of Sorenson [Sorenson 74, Sorenson & Hamacher 75].

The following theorem establishes necessary conditions for ensuring the correctness of any non-preemptive discipline that does not use inserted idle time. For convenience, we assume throughout this section that our set of tasks is sorted in non-decreasing order by period (if $i > j$, then $p_i \geq p_j$). The *index* of a task refers to its position in this sorted list.

Theorem 2.1: Let τ be a set of periodic tasks $\{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by period. τ can be scheduled non-preemptively on a uniprocessor without inserted idle time for all possible release times, only if:

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

2) $\forall i, 1 < i \leq n; \forall L, p_1 < L < p_i: L \geq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j$.

Informally, condition (1) can be thought of as a requirement that the processor not be overloaded. If a periodic task T has a cost c and period p, then $\frac{c}{p}$ is the fraction of

processor time consumed by T over the lifetime of the system (i.e., the utilization of the processor by T). The first condition simply stipulates that the cumulative processor utilization cannot exceed one.[2]

The right hand side of the inequality in the second condition is the worst case processor demand that can be realized in an interval of length L starting at the time an execution request of a task T_i is scheduled and ending sometime before the time the execution request must complete. For a set of tasks to be feasible, the demand in this interval must always be less than or equal to the length of the interval. Although this is semantically similar to a utilization requirement, we will later show that conditions (1) and (2) are in fact not related.

Proof: To show that these conditions are necessary for all possible release times, we need only demonstrate that there exist release times for which conditions (1) and (2) are necessary for τ to be feasible. We first show that condition (1) is necessary.

For a set of tasks τ, the *processor demand* in the interval $[a,b]$, written $d_{a,b}$, is defined as the minimal processing time required by τ in the interval $[a,b]$. That is, $d_{a,b}$ is the minimum amount of processor time required in the interval $[a,b]$ to ensure that no deadline is missed in the interval $[a,b]$. If a set of tasks τ is feasible, then for all a and b, $a < b$, it follows that $d_{a,b} \leq b - a$.

For all i, $1 \leq i \leq n$, let $s_i = 0$ and let $t = p_1 \cdot p_2 \cdot \ldots \cdot p_n$. In the interval $[0,t]$, $\frac{t}{p_i} c_i$ is the total processor time that must be allocated to task T_i to ensure that T_i does not miss a deadline in the interval $[0,t]$. If τ is feasible then

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

or simply

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

---

[2] In [Liu & Layland 73] it was shown that if all tasks are released at time zero, then condition (1) is necessary and sufficient for scheduling a set of periodic tasks on a uniprocessor with the preemptive EDF algorithm.

For condition (2), choose a task T_i, $1 < i \leq n$, and let $s_i = 0$, $s_j = 1$ for $1 \leq j \leq n$, $j \neq i$. This gives rise to the pattern of task execution requests shown in Figure 2.1 below.

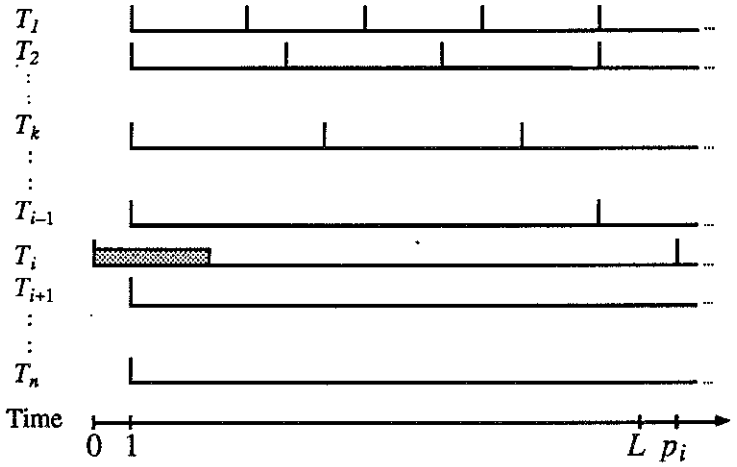


Figure 2.1: Construction for the necessity of condition (2).

For all L, $p_1 < L < p_i$, in the interval $[0, L]$, the processor demand, $d_{0,L}$, is given by

$$d_{0,L} = c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j .$$

Hence for τ to be feasible we must have

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

Conditions (1) and (2) from Theorem 2.1 are also necessary for the correct, non-preemptive scheduling of a set of sporadic tasks when inserted idle time is disallowed. This follows immediately from the previous theorem and our definition of sporadic tasks.

Corollary 2.2: A set of sporadic tasks $\tau = \{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by period, can be scheduled non-preemptively without inserted idle time for all possible release times only if τ satisfies conditions (1) and (2) from Theorem 2.1.

Proof: Based on our definitions of periodic and sporadic tasks, a periodic task is a special case of a sporadic task. Therefore, any conditions necessary for scheduling a set of periodic tasks must also be necessary for scheduling a set of sporadic tasks. By Theorem 2.1, a set of periodic tasks can be scheduled non-preemptively without inserted idle time for all possible release times only if conditions (1) and (2) hold, hence the same must hold for sporadic tasks. □

We next demonstrate the optimality of the non-preemptive EDF discipline over all non-preemptive disciplines that do not use inserted idle time for scheduling sporadic tasks for all possible release times. This means if any non-preemptive algorithm that does not use inserted idle time can correctly schedule a set of sporadic tasks for all possible release times, then the non-preemptive EDF algorithm will also correctly schedule the tasks. To prove optimality, it suffices to show that conditions (1) and (2) are sufficient for ensuring the correctness of the non-preemptive EDF algorithm.

Theorem 2.3: Let τ be a set of sporadic tasks $\{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by period. τ can be scheduled by the non-preemptive EDF discipline for all possible release times, if conditions (1) and (2) from Theorem 2.1 hold.

Proof: (By contradiction.)

Assume the contrary, i.e., that conditions (1) and (2) from Theorem 2.1 hold and yet there exists a set of release times such that a task misses a deadline at some point in time when τ is scheduled by the non-preemptive EDF algorithm. The proof proceeds by deriving upper bounds on the processor demand for an interval ending at the time a task misses a deadline.

Let t_d be the earliest point in time at which a deadline is missed. The set of sporadic tasks can be be partitioned into three disjoint subsets:

S_1 = the set of tasks that have an execution request interval with a deadline at time t_d,

S_2 = the set of tasks that have an execution request interval that contains the point t_d as an interior point, or

S_3 = the set of tasks not in S_1 or S_2.

Tasks in S_3 either have a release time greater than t_d, or they have ceased making requests for execution immediately prior to time t_d. To bound the processor demand prior to t_d, it

suffices to concentrate on the tasks in S_2. Let $b_1, b_2, ..., b_k$ be the request times immediately prior to t_d of the tasks in S_2. There are two cases to consider.

Case 1: None of the execution requests made at the times $b_1, b_2, ..., b_k$ by tasks in S_2 are scheduled prior to 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 cumulative processor demand is the total processing requirement of tasks that male requests for execution at or after time t_0, and that have deadlines at or before time t_d. This gives

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

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

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

However, this is a contradiction of condition (1). Therefore, if conditions (1) and (2) hold and the non-preemptive EDF discipline fails to schedule τ, then some of the execution requests of tasks with request intervals containing the point t_d must have been scheduled prior to t_d.

Case 2: Some of the execution requests made at the times $b_1, b_2, ..., b_k$ by tasks in S_2 are scheduled prior to t_d.

Let T_i be the last task in S_2 scheduled prior to t_d. Let t_i be the point in time at which the request interval of T_i that contains t_d commences execution.

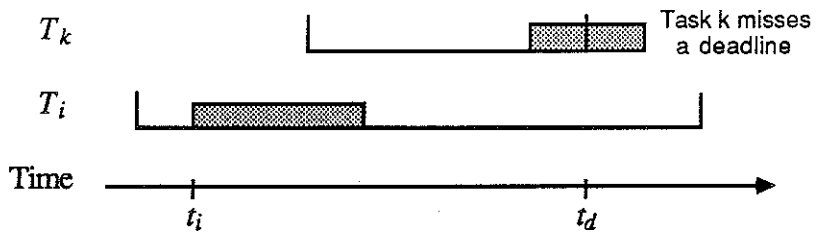


Figure 2.2: Construction for the sufficiency of condition (2).

We will show that if the request 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 $[t_i, t_d]$ to schedule all request intervals of tasks with deadlines at or before time t_d. To begin, we derive an upper bound on the processor demand for the interval $[t_i, t_d]$.

The following facts hold for Case 2:

i) $p_i > t_d - t_i$.

Follows immediately from the definition of task T_i.

ii) No task with index greater than i is scheduled in the interval $[t_i, t_d]$.

Since task T_i is the last task in S_2 scheduled prior to t_d, every other task scheduled in $[t_i, t_d]$ has a deadline at or before t_d. Since any task with an index greater than i has a period at least as big as p_i, if a task T_j, $j > i$, is scheduled in the interval $[t_i, t_d]$, then the T_j must have been available for execution at time t_i. Consequently, since task T_i has a deadline after t_d, the EDF algorithm will always choose task T_j before T_i in the interval $[t_i, t_d]$. Therefore, no task with index greater than i is scheduled in the interval $[t_i, t_d]$.

iii) Other than task T_i, no task which is scheduled in $[t_i, t_d]$ could have made a request for execution at t_i.

Again, as a consequence of the definition of task T_i, other than T_i, every task scheduled in $[t_i, t_d]$ has a deadline at or before t_d. Therefore, if a task $T_{i'}$, that is scheduled in $[t_i, t_d]$ had made a request for execution at t_i, the non-preemptive EDF discipline would have scheduled task $T_{i'}$ instead of task T_i at time t_i.

iv) 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 $t_0 > t_i$ is the end of the last idle period prior to time t_d, to reach a contradiction of condition (1).

Fact (*ii*) indicates that only tasks T_1 - T_i need be considered in computing d_{t_i,t_d}. Since the request interval of task T_i scheduled at time t_i has a deadline after time t_d, all outstanding requests for execution at t_i with deadlines at or before t_d must have been satisfied by t_i. Therefore, when computing d_{t_i,t_d}, we need not consider request intervals of tasks T_1 - T_{i-1} that contain the point t_i as an interior point. Similarly, since T_i has the last execution request with deadline after t_d scheduled prior to t_d, the request intervals of tasks T_1 - T_{i-1} that contain the point t_d as an interior point need not be considered. Lastly, since none of the tasks T_1 - T_{i-1} made a request for execution at time t_i, the 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 indicate that the cumulative processor demand in $[t_i, t_d]$ is bounded by

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

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 (*iv*) indicates that there is no idle time in $[t_i, t_d]$, and since a task missed a deadline at t_d, it follows that $t_d - t_i < d_{t_i,t_d}$. Since $L = t_d - t_i$, this is simply $L < d_{t_i,t_d}$. Combining this with the inequality above yields

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

Since from (*i*), $p_i > t_d - t_i$, we have $L < p_i$. Furthermore, since a task missed a deadline at time t_d, there must have been a task with an execution request wholly contained within the interval $[t_i, t_d]$. Therefore it must be the case that $t_d - t_i > p_1$, or $L > p_1$. Therefore the above inequality contradicts the fact that condition (2) was assumed to be true. This concludes Case 2.

We have shown that in all cases, if the non-preemptive EDF algorithm fails then either condition (1) or condition (2) must have been violated. This proves the theorem. □

Theorem 2.3 has shown that with respect to the class of scheduling policies that do not use inserted idle time, the non-preemptive EDF scheduling discipline is an optimal non-preemptive discipline for sporadic tasks when all possible release times are considered. The following corollary shows that the non-preemptive EDF discipline is also optimal for a set of periodic tasks for all possible release times. This again follows immediately from the previous theorem and our definition of a sporadic task.

Corollary 2.4: Let τ be a set of periodic tasks $\{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by period. The non-preemptive EDF discipline is correct with respect to τ for all possible release times if conditions (1) and (2) from Theorem 2.1 hold.

Proof: Recall that a periodic task is a sporadic task. Therefore, if conditions (1) and (2) are sufficient for guaranteeing the correctness of the non-preemptive EDF discipline when scheduling a set of sporadic tasks for all possible release times, then conditions (1) and (2) are also sufficient for guaranteeing the correctness of the non-preemptive EDF discipline when scheduling a set of periodic tasks for all possible release times. □

We can in fact show a stronger optimality result for sporadic tasks. Based on our definition of a sporadic task, we can show that the non-preemptive EDF discipline is an optimal discipline for scheduling sporadic tasks with arbitrary release times.

Lemma 2.5: A set of sporadic tasks τ, can be feasible for an arbitrary set of release times only if it is feasible for all possible release times.

Proof: By the definition of sporadic tasks, an arbitrary amount of time may elapse 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 such that 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 length 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. □

Theorem 2.6: With respect to the class of scheduling algorithms that do not use inserted idle time, the non-preemptive EDF discipline is an optimal discipline for scheduling sporadic tasks with arbitrary release times.

Proof: The proof follows immediately from Theorem 2.3 and Lemma 2.5. □

2.2 Complexity of Deciding Feasibility For Sporadic Tasks

Since the non-preemptive EDF discipline is optimal for sporadic tasks, in order to decide if a set of sporadic tasks is feasible on a uniprocessor, one need only consider if conditions (1) and (2) from Theorem 2.1 hold. Deciding if condition (1) holds is straightforward and can be performed in time $O(n)$. In this section we give an $O(p_n)$ decision procedure for determining if condition (2) holds. (Recall that p_n is the period of the "largest" task.)

Let

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

Intuitively, $f(L)$ is the processor demand for the interval $[0, L-1]$ when all tasks are released at time zero. For all L, $p_1 < L < p_n$, we can compute $f(L)$ in time $O(p_n)$ as follows. Initialize an array of integers A of size p_n to zero. For each task T_k, $1 \le k \le n$, add c_k to location m of array A for all m that are multiples of p_k. At the completion of this process the sum of the first $l-1$ locations of A will be $f(l)$. 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 total number of execution requests that must be completed before time $p_n - 1$ when all tasks are released at time zero. Note that if a set of tasks satisfies condition (1) then the latter term can be at most p_n. In addition, note that the array A need only be of size $p_n - p_1$ since for all l, $0 \le l < p_1$, $f(l) = 0$. However, this optimization does not effect the time complexity of the computation.

For $p_1 < p < p_n$, let

$$M(p) = \underset{p_1 < L < p}{\text{MIN}}(L - f(L)) .$$

Intuitively, $M(p)$ is the minimum amount of time the processor will have sat idle over all times $L < p$ if all tasks with periods less than p are released at time one (and all tasks with period greater than or equal to p are released after time p). For all p, $p_1 < p < p_n$, the time required for computing $M(p)$ is again $O(p_n)$.

Since for all $l < p_i$,

$$\sum_{j=1}^{i-1} \left\lfloor \frac{l}{p_j} \right\rfloor c_j = \sum_{j=1}^{n} \left\lfloor \frac{l}{p_j} \right\rfloor c_j$$

for all i, $1 < i \leq n$, a set of tasks will satisfy condition (2) if and only if for each i, $1 < i \leq n$, $M(p_i) \geq c_i$. This final determination can be made in time $O(n)$. For all task sets of size n which satisfy condition (1), it follows that $p_n \geq n$. Therefore, the time required to the decide feasibility of a set of sporadic tasks is dominated by the time required to compute $f(L)$; namely $O(p_n)$.

There are two possible variations on this strategy. The first will be faster if the number of execution requests made in the interval $[0,p_n]$ when all tasks are released at time zero is much smaller than p_n (for example, if $p_n - p_1$ is small).

The function $f(L)$ can also be computed by simulating the earliest deadline algorithm in the interval $[0,p_n]$ and summing the processing times of the requests. If R is the total number of execution requests made up to time p_n, then the total time required to compute $f(L)$ will be $R\log n$. This method also reduces the space needed to $O(n)$. A second way to reduce the space to $O(n)$ is to compute the array A in windows of size n. For each task we compute its multiples until the next one is outside the current window. We then update a cumulative sum of all the $A[j]$ values computed so far, and test any p_i values which fall within the current window. The total time required per window is $O(n)$, hence the total time is still $O(p_n)$. This last analysis assumes that $p_n \geq n^2$, otherwise the computation will be $O(n^2)$.

Note that in several cases the time complexity bound 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 technically 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 the task descriptions that are most likely to be encountered in practice, one can efficiently determine the feasibility of the tasks. Lastly, we mention that it is not known if condition (2) can be evaluated in time polynomial in n alone. Therefore it is possible that the true complexity of evaluating condition (2), and hence the complexity of deciding the feasibility of a set of sporadic tasks, is NP-complete.

2.3 Scheduling With Inserted Idle Time

All of our optimality results for non-preemptive scheduling have been with respect to the class of non-preemptive disciplines that do not use inserted idle time. That adopting inserted idle time yields a more powerful scheduling algorithm can be seen by the following example. Consider the two periodic tasks (recall T = (release time, cost, period)):

$$T_1 = (9, 8, 20), \text{ and}$$
$$T_2 = (0, 23, 40).$$

These tasks cannot be scheduled correctly using any non-preemptive discipline that does not use inserted idle time. Any such discipline would necessarily schedule T_2 at time 0 and T_1 at time 23 as shown in the simulation depicted in Figure 2.3.

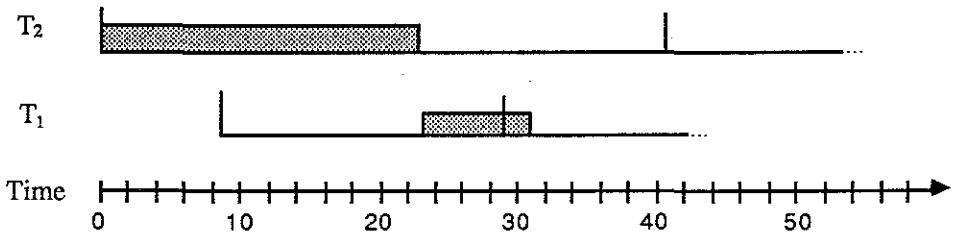


Figure 2.3: Scheduling without inserted idle time.

These choices of release times force T_1 to miss a deadline at time 29 when inserted idle time is disallowed. These two tasks can, however, be scheduled correctly by a non-preemptive algorithm that idles the processor as shown in Figure 2.4.

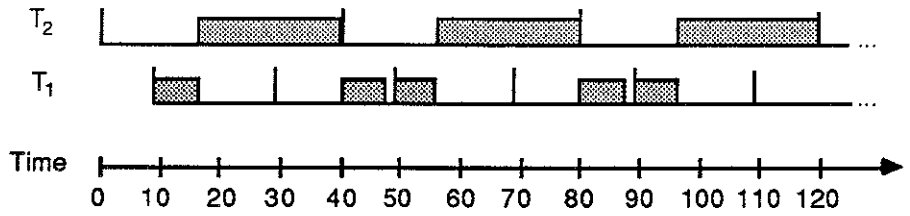


Figure 2.4: Scheduling with inserted idle time.

We discuss our emphasis on scheduling without inserted idle time in greater detail in Section 4.

3. Non-Preemptive Scheduling of Periodic Tasks

For periodic tasks, we limited our attention in Section 2 to scheduling problems where all possible release times were considered. The existence of an optimal scheduling discipline for periodic tasks is quite sensitive to knowledge of the release times. This section provides evidence that there may not exist an efficient optimal non-preemptive uniprocessor scheduling discipline for periodic tasks with arbitrary release times.

To begin, the definition of optimality presented in Section 1 must be refined to include some notion of efficiency. So far, it has been assumed that a scheduling discipline can determine which task to schedule next in zero time. Under this assumption, a scheduler that enumerated all possible schedules would be an optimal, albeit uninteresting, scheduler. Therefore, in addition to correctly scheduling all task sets that are feasible on a uniprocessor, an optimal scheduling discipline should be required to make each scheduling decision in time polynomial in the number of tasks. With this new notion of optimality, we will show that if there exists an optimal non-preemptive scheduling discipline for scheduling periodic tasks with arbitrary release times on a uniprocessor, then P = NP.

The following theorem shows that the complexity of deciding if a set of periodic tasks is feasible on a uniprocessor when one is allowed to consider any non-preemptive scheduling discipline (including those that allow inserted idle time) is NP-hard in the strong sense. This means that unless P = NP, a pseudo-polynomial time algorithm does not exist for deciding feasibility of periodic tasks with arbitrary release times [Garey & Johnson 79]. This provides strong evidence that the problem is intractable. This decision problem can be formally stated as follows.

NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS: Given a set of periodic tasks $\tau = \{T_i = (s_i, c_i, p_i) \mid 1 \leq i \leq n\}$ with $s_i, c_i, p_i \in \mathbf{Z}^+$, is it possible to correctly schedule τ non-preemptively, on a uniprocessor?

Theorem 3.1: NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS is NP-hard in the strong sense.

Proof: We will give a polynomial time transformation from the 3-PARTITION problem ([Garey & Johnson 79]) to NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS.

An instance of the 3-PARTITION problem consists of a finite set A of $3m$ elements, a bound $B \in \mathbf{Z}^+$, and a "size" $s(a) \in \mathbf{Z}^+$ for each $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$, and $\sum_{j=1}^{3m} s(a_j) = Bm$. The problem is to determine if A can be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ such that, for $1 \leq i \leq m$, $\sum_{a \in S_i} s(a) = B$. (With the above constraints on the element sizes, note that every S_i will contain exactly three elements from set A.)

The transformation is performed as follows. Let $A = \{a_1, a_2, a_3, \ldots, a_{3m}\}$, $B \in \mathbf{Z}^+$, and $s(a_1), s(a_2), s(a_3), \ldots, s(a_{3m}) \in \mathbf{Z}^+$, constitute an arbitrary instance of the 3-PARTITION problem. We create an instance of NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS by constructing a set τ of $n = 3m + 2$ periodic tasks as follows:

$$\tau = \{\ T_1 = (0, 8B, 20B),$$
$$T_2 = (9B, 23B, 40B),$$

$$\forall j,\ 3 \leq j \leq 3m+2:\ T_j = (0, s(a_{j-2}), 40Bm)\ \}.$$

This construction can clearly be done in polynomial time with the largest number created in the new problem instance being $40Bm$. In this instance of NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS, note that the system utilization is

$$\sum_{j=1}^{n} \frac{c_j}{p_j} = \frac{8}{20} + \frac{23}{40} + \frac{\sum_{j=1}^{3m} s(a_j)}{40Bm} = \frac{39}{40} + \frac{Bm}{40Bm} = 1.$$

By our choice of release times for T_1 and T_2, τ can be feasible under a non-preemptive scheduling discipline only if T_2 is scheduled at points in time $9B + 40Bk$, and all the

request intervals of T_1 that begin at time $20B + 40Bk$, are scheduled at time $40B(k+1) - 8B$, for all $k \geq 0$. Such an execution of τ is shown in Figure 3.1 below (with the execution of the requests of task T_1 made at times $20Bk$ omitted for clarity).

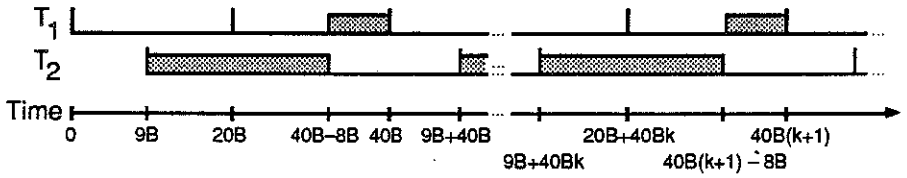


Figure 3.1: The only feasible execution schedule for tasks T_1 and T_2.

If the i^{th} request interval of T_2 is scheduled at some time other than $9B + 40B(i-1)$, then the request interval of T_1 made at time $20B + 40B(i-1)$ will miss its deadline. Similarly, if a request interval of T_1 made at time $20B + 40Bk$ for some k, $k \geq 0$, is scheduled at some time other than at $40B(k+1) - 8B$, then the request interval of T_2 made at time $9B + 40Bk$ will miss its deadline.

Note that with these scheduling constraints, if we consider only tasks T_1 and T_2, then for all i, $i > 0$, in each interval $[40B(i-1), 40Bi]$, the processor will be idle for exactly B time units. It follows that in the interval $[0, 40Bm]$, there will be I separate idle periods, $m \leq I \leq 2m$, whose total duration is exactly Bm time units.

For example, Figure 3.2 below depicts a simulation of the non-preemptive EDF discipline on the tasks in τ. With the non-preemptive EDF discipline in the interval $[0, 40Bm]$ there will be exactly m separate idle periods, each of duration B time units. For this policy, τ will be feasible if and only if the EDF algorithm can schedule tasks in T_3 - T_{3m+2} in these m idle periods.

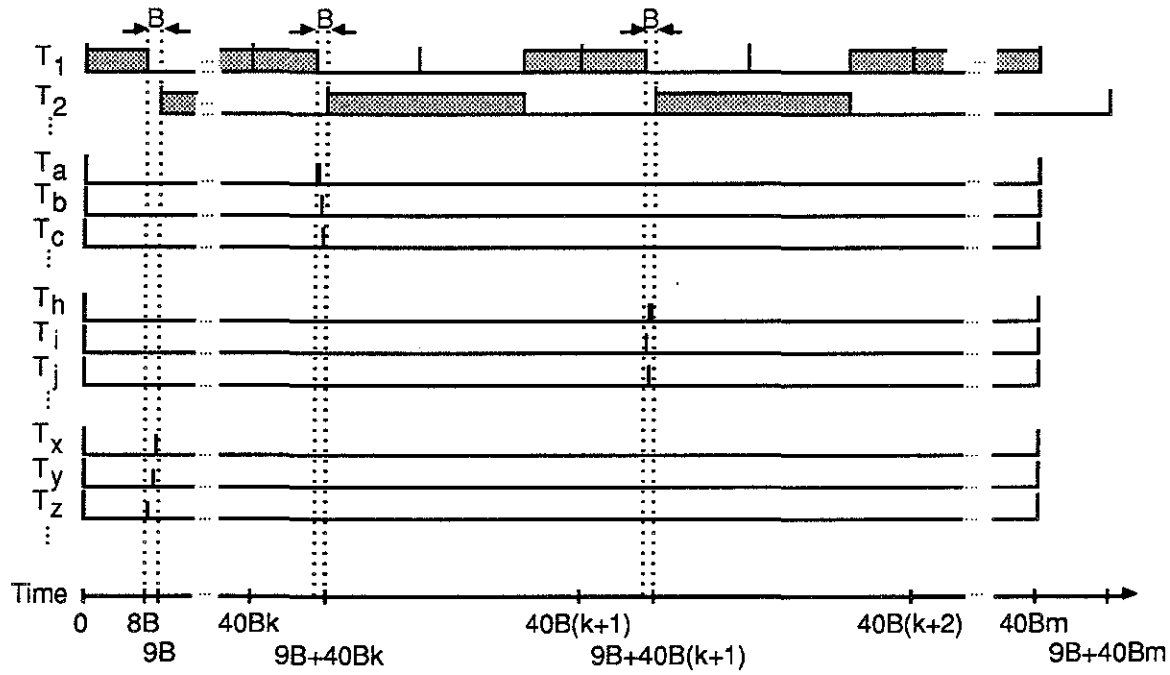


Figure 3.2: Execution of τ under the non-preemptive EDF scheduling discipline.

In the general case, τ will be feasible on a uniprocessor under a non-preemptive scheduling discipline if and only if there exists a partition of tasks T_3 - T_{3m+2} into m disjoint sets S_1, S_2, ... , S_m, such that for each set S_i, $\sum_{T_j \in S_i} c_j = B$.

Therefore a solution to NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS can be used to solve an arbitrary instance of the 3-PARTITION problem by simply constructing a set of periodic tasks as shown in the beginning of this proof, and then presenting this set of tasks to a decision procedure for NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS. The answer from the NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS decision procedure is the answer to the 3-PARTITION question for this problem instance. Since 3-PARTITION is known to be NP-complete in the strong sense [Garey & Johnson 79], NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS must be at least NP-hard in the strong sense. □

The situation is actually bleaker for periodic tasks. The construction of τ in Theorem 3.1 can also be used to show that if an optimal non-preemptive uniprocessor scheduling discipline existed for scheduling periodic tasks, and this discipline took only a polynomial amount of time (in the length of the input) to make each scheduling decision, then P = NP.

That is, if there exists an optimal non-preemptive scheduling discipline for periodic tasks, then we can give a pseudo-polynomial time algorithm for deciding 3-PARTITION. The key observation is that if a 3-PARTITION problem instance is embedded in NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS as described above, then only a pseudo-polynomial length portion of the schedule generated by an optimal non-preemptive discipline when scheduling τ, needs to be checked in order to decide the embedded 3-PARTITION problem instance.

Corollary 3.2: If there exists an optimal, non-preemptive, uniprocessor scheduling discipline for scheduling periodic tasks then P = NP.

Proof: Assume there exists such an optimal scheduling discipline. From an instance of the 3-PARTITION problem, construct a set τ of periodic tasks as described in the proof of Theorem 3.1. Note that if τ is not feasible, then some task in τ will miss a deadline in the deadline in the interval $[0, 9B+40Bm]$. Therefore we can simulate the optimal scheduling discipline on τ over the interval $[0, 9B+40Bm]$ and simply check to see if any tasks miss a deadline in this interval. The simulation and the checking of the schedule produced by the optimal discipline can clearly be performed in time proportional to $40Bm$. By the reasoning employed in the proof of Theorem 3.1, if some task missed a deadline then there is a negative answer to the 3-PARTITION problem instance. If no task missed a deadline then there is an affirmative answer. Therefore, since 3-PARTITION is NP-complete in the strong sense and since we have given a pseudo-polynomial time algorithm for deciding 3-PARTITION, P = NP. □

Unless P = NP, Corollary 3.2 shows that we will not be able to develop an optimal non-preemptive scheduling discipline for scheduling periodic tasks with arbitrary release times.

4. Discussion

Condition (1) of Theorem 2.1 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. While condition (2) of Theorem 2.1 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 periodic or sporadic tasks 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. One special case task set worth mentioning arises when all tasks have the same period. In this case condition (2) of Theorem 2.1 is vacuous and the tasks will be feasible if and only if they do not overload the processor.

Condition (2) of Theorem 2.1 expressed the feasibility of a task set in terms of the worst case processor demand that can occur in an interval of length L. For periodic tasks, the intractability of deciding feasibility arises from our inability to efficiently determine if such an interval can ever be realized. This is not an issue for sporadic tasks. Given the potentially non-deterministic behavior of sporadic tasks, we were able to argue that there can always exist an interval wherein the processor demand is given by the right hand side of condition (2).

The non-determinism allowed in the behavior of sporadic tasks has influenced this work in other dimensions as well. Our focus in this paper has been on the on-line scheduling of tasks. This is because it will not be possible to generate a schedule off-line if the execution request 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 make their next requests for execution. In general, this will not be possible for sporadic tasks. Note that while an on-line algorithm for scheduling sporadic tasks clearly cannot use inserted idle time effectively, however it is possible that an off-line scheduling algorithm could be employed for both periodic and sporadic tasks. In general, we do not view off-line scheduling as a viable option as the time and space complexity of constructing off-line schedules, independent of the use of inserted idle time, is likely to be prohibitive. In order for a schedule to have finite length it must be repetitive. For a more refined model of periodic tasks, Leung and Merrill have shown that a repetitive schedule always exists for feasible task sets when preemption is allowed at arbitrary points. However the length of this schedule has an upper bound proportional to the least common multiple of the periods of the tasks [Leung & Merrill 80]. In general, it will take exponential time and space to

generate this schedule. For our tasking model it is not known if this bound can be improved.

In [Jeffay 89a], it was shown that in terms of feasibility, when preemption is allowed at arbitrary points, sporadic and periodic are equivalent characterizations of the repetitive behavior of a real-time process. Our present results have shown that this is not the case when preemption is disallowed. There are some other interesting observations to be made concerning the non-preemptive scheduling of repetitive tasks. In the *preemptive* scheduling domain, the space of scheduling policies is quite "dense" in optimal scheduling policies. For example, in addition to the EDF discipline, an alternate uniprocessor scheduling policy based on the *laxity* of each task has been shown to lead to an optimal preemptive scheduling discipline for periodic tasks. The laxity of a task is defined as the difference between the time remaining until the next deadline of an execution request of a task, and the amount of processor time required to complete the current execution request of the task. That is, if at time t a task has an outstanding request for execution with a deadline at time t_d, then the request's laxity is $t_d - t - c_t$, where c_t is the amount of computation remaining at time t. The laxity of a task indicates the amount of time an execution request of a task can wait before it must be scheduled. For this reason laxity is a better measure of the "time criticalness" of an execution request of a task. For example if we consider two periodic tasks

$$T_1 = (0,1,5), \text{ and}$$
$$T_2 = (0,5,7),$$

then task T_1 has a nominal laxity of 4 and task T_2 has a nominal laxity of 2. That is, an execution request of task T_1 can afford to wait for 4 time units before it must be scheduled while an execution request of task T_2 can afford to wait for only two time units. In this example, a *least laxity first* (LLF) scheduler would schedule task T_2 at time 0. (An EDF scheduler would schedule task T_1 at time 0.)

Dertouzos has shown that an LLF scheduling discipline is an optimal preemptive discipline for periodic tasks [Dertouzos 74]. In fact Mok has remarked that there exist an infinite number of optimal preemptive uniprocessor scheduling disciplines for periodic tasks [Mok 83]. In essence, any scheduler that alternates between EDF and LLF scheduling can be shown to be optimal. We mention these facts only to point out the LLF scheduling discipline is *not* an optimal non-preemptive scheduling discipline. This can be seen by

noting that the tasks in the example above are feasible for arbitrary release times since they satisfy the conditions of Theorem 2.1. However under a non-preemptive LLF scheduler, the first execution request of task T_1 will miss a deadline at time 5. This provides some preliminary evidence that the number of plausible non-preemptive scheduling policies may indeed be quite limited.

5. Summary

In summary, this paper has demonstrated the following key results. For non-preemptive scheduling, the EDF discipline is optimal for sporadic tasks and for periodic tasks when all possible release times are considered. The optimality is with respect to the class of scheduling policies that do not use inserted idle time. Unless P = NP, there does not exist an optimal non-preemptive scheduling discipline for periodic tasks with arbitrary release times. Table 1 below summarizes these results. Table 2 gives the complexity measures for deciding feasibility. The important aspects of these results are:

- sporadic and periodic tasks are not equivalent in terms of feasibility,
- for sporadic tasks, feasibility is not dependent on knowledge of release times,
- for periodic tasks, feasibility is dependent on knowledge of release times,
- feasibility of sporadic tasks can be determined efficiently for tasks with bounded periods, and
- the problem of determining the feasibility of a set of periodic tasks with arbitrary release times is intractable.

Table 1: Optimal non-preemptive scheduling disciplines.

	Arbitrary Release Times	All Possible Release Times
Sporadic Tasks	Non-preemptive EDF	Non-preemptive EDF
Periodic Tasks	If a polynomial time algorithm algorithm exists, then P = NP	Non-preemptive EDF

Table 2: Complexity of deciding feasibility when preemptive is disallowed.

	Arbitrary Release Times	All Possible Release Times
Sporadic Tasks	Pseudo-polynomial time $O(p_n)$	Pseudo-polynomial time $O(p_n)$
Periodic Tasks	NP-hard in the strong sense	Pseudo-polynomial time $O(p_n)$

Although there may not exist an optimal algorithm for scheduling periodic tasks with arbitrary release times, Corollary 2.4 has established sufficient conditions for ensuring the correctness of the non-preemptive EDF discipline. This is certainly useful for problems such as scheduling periodic tasks when the release times are unknown.

5. Acknowledgments

We would like to thank Alan Shaw, Ewan Tempero, and John Zahorjan for their comments on earlier drafts of this paper.

6. References

[Bertossi & Bonuccelli 83]
Bertossi, A.A., Bonuccelli, M.A., *Preemptive Scheduling of Periodic Jobs in Uniform Multiprocessor Systems*, **Information Processing Letters**, Vol. 16, No. 1, (January 1983), pp. 3-6.

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

[Dhall & Liu 78]
Dhall, S.K., Liu, C.L., *On a Real-Time Scheduling Problem*, **Operations Research**, Vol. 26, No. 1, (January 1978), pp. 127-140.

[Dertouzos 74] Dertouzos, M.L., .K., *Control Robotics: The Procedural Control of Physical Processes*, Proc. of the IFIP Congress, August 1974, Stockholm, Sweden, pp. 807-813.

[Frederickson 83]
Frederickson, G.N., *Scheduling Unit-Time Tasks with Integer Release Times and Deadlines*, **Information Processing Letters**, Vol. 16, No. 4, (May 1983), pp. 171-173.

[Garey & Johnson 79]
Garey, M.R., Johnson, D.S., **Computing and Intractability, A Guide to the Theory of NP-Completeness**, W.H. Freeman and Company, New York, 1979.

[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.

[Lawler & Martel 81]
Lawler, E.L., Martel, C.U., *Scheduling Periodically Occurring Tasks on Multiple Processors*, **Information Processing Letters**, Vol. 12, No. 1, (February 1981), pp.9-12.

[Leung & Merrill 80]
Leung, J.Y.-T., Merrill, M.L., *A Note on Preemptive Scheduling of Periodic, Real-Time Tasks*, **Information Processing Letters**, Vol. 11, No. 3, (November 1980), pp.115-118.

[Leung & Whitehead 82]
Leung, J.Y.-T., Whitehead, J., *On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks*, **Performance Evaluation**, Vol. 2, No. 4, (1982), pp.237-250.

[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.

[Sorenson 74]
Sorenson, P.G., *A Methodology for Real-Time System Development*, Ph.D. Thesis, University of Toronto, June 1974.

[Sorenson & Hamacher 75]
Sorenson, P.G., Hamacher, V.C., *A Real-Time Design Methodology*, **INFOR**, Vol. 13, No. 1, (February 1975), pp. 1-18.