A Foray into Uniprocessor Real-Time Scheduling Algorithms and Intractibility

Ed Overton

Drs. T. Brylawski [1] and J. Anderson [2], advisors

December 3, 1997

[1]UNC-CH Department of Mathematics
[2]UNC-CH Department of Computer Science

Contents

I would like to thank Drs. Brylawski and Anderson for their continued efforts to help me with this paper. In particular, I thank Dr. Brylawski for his flexibility in working with me, and I thank Dr. Anderson for his time and energy working with a student outside of his department.

1 Introduction

Scheduling theory may be thought of as the study of how to accomplish certain tasks by certain deadlines. As humans, we handle scheduling issues every day. For example, a student must accomplish homework by the appropriate due date, a professor must complete the rought draft of a paper by the submission date, etc. Were we to have only one task to accomplish, meeting that deadline probably would be very simple. Our lives, however, contain many tasks that have deadlines – tax forms, car inspections, meetings, classes, etc. Thus, we must use some sort of scheduling technique to "juggle" our various tasks, so that they all are completed by their appropriate deadlines. Clearly, there are some tasks whose deadlines are not strict – one may request an extension to submit one's taxes, one may balance the chance of getting ticketed against missing one's car inspection date, etc. However, there are those tasks in life whose deadlines are much more strict – court dates, grant proposals, etc. In such situations, if the deadline is missed, there are dire consequences (e.g., being sent to jail, not receiving funding). These examples show that we live life in real time: a situation where homework and grant proposals must be properly submitted, but also one where they must be submitted on time.

In this paper, we will consider hard-real-time systems. A real-time system is one where computations not only must produce correct output, but also must produce that output in a timely fashion (namely, by given deadlines). Hard-real-time systems are real-time systems where the cost associated with untimely output (i.e., missing a deadline) is very high. An example of a hard-real-time system is a computer that controls the landing of an airplane: the rudders and flaps must respond to the sensors' inputs within a given timeframe. If the responses are incorrect or are too late, the plane may crash. Due to the nature of hard-real-time systems, consideration is focused primarily (and in this paper, focused solely) on worst-case behavior. If a scheduling system produces wonderful output in the average case, but is known to fail in some situations, one would not wish to trust such a system to landing an airplane. Clearly, one would desire a guarantee of a correct landing.

Hard-real-time systems are usually considered as a set of tasks that are repeatedly requested. Tasks may be *periodic*, where there is a constant amount of time t such that the task is requested every t time units (e.g., a digital watch changing its display every second), or *sporadic*, where each task request must arrive at least a constant time after the previous request (e.g., resetting the time on a digital watch). In this paper, we will focus on periodic tasks, and discuss the effects of considering sporadic tasks where appropriate. Either way, each task request has an associated deadline, by which the task must complete execution. Additionally, the tasks may have *shared resources* – objects which some tasks require (exclusively) at some point during execution. An example of a shared object would be a computer's hard drive – one program may wish to write its data while the other wishes to read data from another location on the disk. Since a disk cannot write from one location and read from another at the same time, the read and write tasks require exclusive access to the disk during the appropriate read and write portions of their execution.

A scheduling algorithm is one that uses the information from the set of tasks, and discerns when to schedule what task. All scheduling algorithms may be considered as priority driven – the task with the highest priority that has execution remaining should be scheduled. In that regard, there are two subsets of scheduling algorithms: ones where priorities are fixed (static priorities), and ones where priorities may change over time (dynamic priorities). We will consider four scheduling algorithms for task sets of periodic tasks without shared resources: rate monotonic [LL '73, LSD '89, La '74], and deadline monotonic [LW '82] use static priorities; earliest deadline first [LL '73, LM '80, BRH '90, BHR '93], and modified least laxity first use dynamic priorities. The modified least laxity first scheduling algorithm was developed by the author to generalize two dynamic priority scheduling algorithms – earliest deadline first, and least laxity first [Mo '83].

We will define each scheduling algorithm, and determine in which situations the algorithm is optimal. We will also derive feasibility tests in order to determine if a given task set will have a valid schedule under a given scheduling algorithm. We will also determine the complexity of the feasibility test. The complexity is a significant factor in using the scheduling algorithms, since it gives a rough idea of the time that is involved with determining *a priori* whether a given task set has a valid schedule under the algorithm. In a hard-real-time setting, one would not wish to simply start up the scheduler and hope for the best – one would want to know with certainty that all deadlines will be met. However, since we are considering hard-real-time systems, time may be critical, and the time it takes to determine feasibility might defer the start of the schedule while feasibility is determined. In such a case, one would clearly want a "quick" feasibility test.

Following the work of [LL '73, LSD '89, La '74], we will determine that for the conditions where rate-monotonic scheduling is optimal, there is a linear time feasibility test for rate montonic scheduling that determines necessity, but not sufficiency. We will show that there exists a pseudo-polynomial time necessary and sufficient test that indicates the feasibility question for rate monontonic scheduling is in *NP*.

We will then follow [LW '82] to show it is a generalization of rate-monotonic scheduling, and to determine the conditions under which deadline-monotonic scheduling is optimal. We will then begin to consider task sets where the tasks are not released simultaneously, and see that there is a necessary and sufficient test for feasibility in this case. The test, however, is then shown to be co-NP-complete in the strong sense.

Earliest-deadline-first scheduling is a very powerful scheduling algorithm, as we shall see. It is optimal among dynamic scheduling algorithms and offers the first polynomial time necessary and sufficient feasibility test for feasibility. However, under certain circumstances that test is not valid, and we will show that the feasibility test for those cases is co-NP-complete in the strong sense.

We will lastly develop a new scheduling algorithm, modified least laxity first. The algorithm will be shown as a generalization of earliest-deadline-first scheduling, and is therefore optimal. Additionally, it then inherits the feasibility tests from earliest-deadline-first – under some conditions, we have a polynomial time test, and under others the test is co-NP-complete in the strong sense.

2 Preliminary definitions and notation

We define a periodic task without resources, τ_i, to be the 4-tuple (e_i, d_i, p_i, r_i) where e_i, d_i, and p_i are positive real numbers, and r_i is a non-negative real number. The task τ_i is said to have execution time e_i, a deadline span of d_i, a period of p_i, and an initial release time of r_i. τ_i is said to have release times at $r_{i,k}$, $k \in \mathbb{Z}_+$, where $r_{i,k+1} = r_i + kp_i$. $r_{i,k}$ is said to be the k^{th} release of τ_i. Each release $r_{i,k}$ has an associated deadline, $r_{i,k} + d_i$, the k^{th} deadline of τ_i. We define a task set of n tasks without shared resources, T, to be $\{\tau_i\}_{i=1}^n$, where $\tau_i = (e_i, d_i, p_i, r_i)$ as above. As a convention in this paper, T will represent a task set, subscripted τ's will represent tasks in T, and n will represent the number of tasks in T.

A *schedule* of T is a function $g : \mathbb{R}_+ \mapsto T \cup \{\emptyset\}$. We say that τ_i is *scheduled at time* t or *on the processor at time* t if $g(t) = \tau_i$, and that the *processor is idle at time* t if $g(t) = \emptyset$. We say a task set is *synchronous* if there exists some r such that for all $i, r_i = r$. Without loss of generality for synchronous task sets, we will also assume that $r = 0$: Given a task set $T = \{\tau_i\}_{i=1}^n$ of n tasks that are synchronous, we define T' to be the set $\{\tau_i'\}_{i=1}^n$ such that $\tau_i' = (e_i, d_i, p_i, 0)$. It is clear that if $g(t)$ is the schedule of T produced by a given scheduling algorithm, then $g'(t)$, the schedule of T' produced by that scheduling algorithm, is exactly $g(t+r)$. Additionally, there will be no task scheduled on $[0, r)$ in g since no task is released until time r. Hence, g is valid if and only if g' is valid.

Given a function $f : A \mapsto B$, and some element $b \in B$, we define

$$\chi_{f,b}(a) = \begin{cases} 0 & : \quad f(a) \neq b \\ 1 & : \quad f(a) = b \end{cases}$$

τ_i is said to be *active at time* t if there exists $k \in \mathbb{Z}_+$ such that $r_{i,k} \leq t < r_{i,k} + d_i$ and $\int_{r_{i,k}}^{t} \chi_{g,\tau_i}(x)\, dx < e$. Informally, the task has been released, but has not completed its execution corresponding to that release. τ_i is said to *overflow* or *miss its deadline* at t if there exists $k \in \mathbb{Z}_+$ such that $t = r_{i,k} + d_i$ (i.e., t is the k^{th} deadline of τ_i) and $\int_{r_{i,k}}^{t} \chi_{g,\tau_i}(x)\, dx < e$. If $\int_{r_{i,k}}^{t} \chi_{g,\tau_i}(x)\, dx \geq e$, we say τ_i *meets its deadline at* t. These definitions correspond to the intuitive notion of a deadline – if the task hasn't executed "enough", then the deadline is missed.

The *response time* of the k^{th} release of a task is the difference between the time the task finishes executing that invocation and the time it was released, which can be seen as the time it takes the task to complete its execution. A *critical instant* of a task (under a given scheduling algorithm) is a release that yields the longest possible response time of that task for the given task set. A schedule is said to be *valid* if all deadlines of all tasks are met. We say that, under a given scheduling algorithm, the *processor is fully utilized* for a given task set if the algorithm produces a valid schedule for the given task set, but an increase in the execution time of any process in the task set would yield an overflow. We call a scheduling algorithm *optimal* if, when there exists a valid schedule for some task set T, then the scheduling algorithm also produces a valid schedule for T.

A *priority-based* scheduling algorithm is one where each task τ_i is assigned a corresponding priority, P_i. These priorities may be either *static* or *dynamic*. Lower priority numbers correspond to higher priorities. That is to say, if $P_1 = 1$ and $P_2 = 2$ then task τ_1 has higher priority than task τ_2. All priority based scheduling algorithms use the following definition for their schedule:

$$g_P(t) = \begin{cases} \tau_i & : \quad \tau_i \text{ is active at time } t \text{ and } \forall j \neq i, P_j < P_i \Rightarrow P_j \text{ is not active.} \\ \emptyset & : \quad \text{there is no active task at time } t \end{cases}$$

Note that if two (or more) active process have the same priority, ties may be broken arbitrarily. Thus, in this paper, for static priority scheduling algorithms, we will assume that no two tasks have the same priority (since one must be chosen over another, and that choice must be static). By convention, we will also assume that for a static priority scheduling algorithm, the task sets under consideration are ordered by priority: $P_1 < P_2 < \ldots < P_n$.

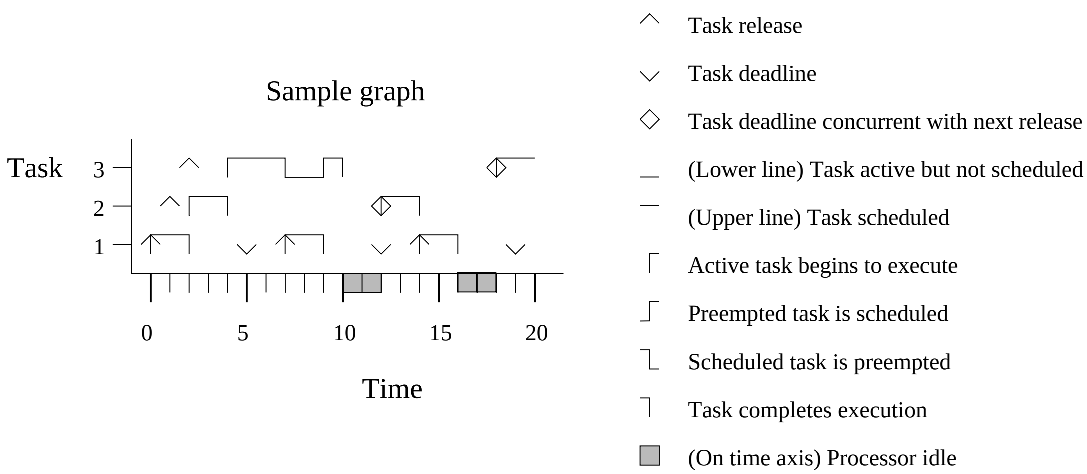
The utilization function corresponds to the notion of "how busy the processor is". Formally,

$U : T \mapsto \mathbb{R}_+$ is defined by

$$U(T) = \sum_{i=1}^{n} \frac{e_i}{p_i}$$

it is clear that $U(T) \geq 0$ for all T, since for all $i, e_i, p_i > 0$. In Section 3 we will show that $U(T) \leq 1$ is a necessary condition to produce a valid schedule of T for any uniprocessor scheduling algorithm.

To facilitate how the scheduling algorithms work, there are several graphs of example task sets. The key to those graphs is as follows:



For example, in the sample graph we have 3 tasks. τ_1 has an execution time of 2, a deadline span of 5, a period of 7, and a release time of 0. τ_2 has an execution time of 2, a deadline span of 11, a period of 11, and a release time of 1. τ_3 has an execution time of 4, a deadline span of 16, a period of 16, and a release time of 2. Thus, at time 0, τ_1 is released and executes. At time 1, τ_2 is released, but is not scheduled since τ_1 is of higher priority. At time 2, τ_1 completes execution and τ_3 is released. Since τ_2 is the higher priority task, it executes to completion at time 4, when τ_3 begins execution. At time 5, τ_1 has a deadline (that is met, since τ_1 finished execution at time 2). At time 7, τ_1 is released and preempts τ_3 until time 9, when τ_3 is again scheduled. τ_3 completes at time 10, when there is no active task. Thus, the processor is idle until a task is released, namely at time 12 (τ_2 is released). The rest of the graph should be clear.

3 Schedulablity and a bound on utilization

We first show one of the fundamental theorems in scheduling theory, which states that no task set with a utilization greater than one is schedulable. Intuitively, this theorem should agree with the reader's notions about utilization – utilization represents the fraction of the time the processor must be active for a valid schedule of the task set. If that fraction is greater than one, then there is more "work" than time available, and the task set has no valid schedule.

We follow the work of [ARJ '97].

Theorem 3.1 ([ARJ '97]) *If a task set is schedulable, its utilization must be at most 1.*

Proof: Assume that there is a valid schedule for some task set T. Then every deadline of the task set will be met. Thus, we know that each task τ_i is scheduled for e_i time units every p_i time units after r_i. Thus, for $t \geq r_i$, τ_i has $\left\lfloor \frac{t-r_i}{p_i} \right\rfloor$ satisfied deadlines over $[r_i, t)$. Note that on $[r_i, t)$, τ_i is then scheduled for $\left\lfloor \frac{t-r_i}{p_i} \right\rfloor e_i$ time units.

So, let $t \in \mathbb{R}_+$ such that $t \geq \max_{\tau_i \in T}\{r_i\}$. Since any valid schedule must meet every deadline, the time available for execution (namely, t) must be greater than the amount of execution corresponding to deadlines at or prior to t:

$$t \geq \sum_{i=1}^{n} \left\lfloor \frac{t - r_i}{p_i} \right\rfloor e_i$$

since $\lfloor x \rfloor > x - 1$ for all $x \in \mathbb{R}_+$, we have

$$\begin{aligned} t &> \sum_{i=1}^{n} \left(\frac{t - r_i}{p_i} - 1 \right) e_i \\ &= \sum_{i=1}^{n} \left(\frac{te_i - r_i e_i}{p_i} - e_i \right) \\ &= t \sum_{i=1}^{n} \left(\frac{e_i}{p_i} \right) - \sum_{i=1}^{n} \left(\frac{r_i e_i}{p_i} + e_i \right) \end{aligned}$$

rearranging terms, we have

$$\sum_{i=1}^{n} \left(\frac{r_i e_i}{p_i} + e_i \right) > t \left(\sum_{i=1}^{n} \left(\frac{e_i}{p_i} \right) - 1 \right)$$

Note that this equation holds for all $t \in \mathbb{R}_+$ such that $t \geq \max_{\tau_i \in T}\{r_i\}$. Since the left-hand side of the equation is a bounded non-negative constant, and the right-hand side is linear in t, we must have

$$\begin{aligned} \left(\sum_{i=1}^{n}\left(\frac{e_i}{p_i}\right) - 1\right) &\leq 0 \qquad \text{, i.e.,} \\ \sum_{i=1}^{n}\frac{e_i}{p_i} &\leq 1 \end{aligned}$$

Thus, if there is a valid schedule for the given task set, then the task set's utilization is at most 1. □

4 Rate Monotonic Scheduling

Rate monotonic scheduling (RM) was the focus of one of the seminal papers in hard-real-time scheduling theory, [LL '73]. The paper laid most of the ground work for much of the development of static-priority scheduling. RM is easy to understand and simple to implement, yet it yields several significant results. Additionally, RM is an optimal scheduling algorithm for static-priority scheduling algorithms under certain circumstances. Due to its significance in the field, we devote a reasonable amount of attention to its development and results.

4.1 Definition

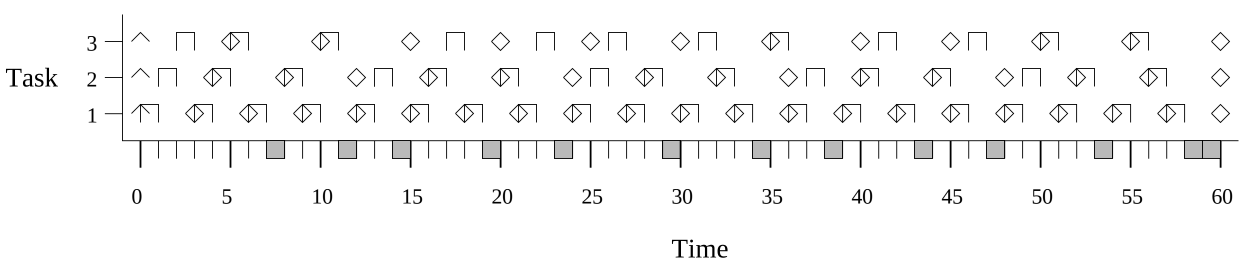
Rate monotonic scheduling is a static-priority scheduling algorithm for periodic tasks. In RM, priorities are equal to the periods of the associated tasks. Hence, the task with the shortest period has the highest priority, and the task with the longest period has the lowest priority. Intuitively, this prioritization makes sense, since the task that has the shortest period will be the first one to be re-released. Hence, it should be the first one to complete (so that it will be ready for its next release). In [LL '73], RM is considered in the case where each task's deadline is concurrent with the task's next release (thus, $d_i = p_i$). [LW '82] later showed that RM is not optimal when this case does not hold, and developed deadline monotonic scheduling (DM). We will consider DM in Section 5. Formally, RM is a priority-based algorithm, such that $P_i = p_i$ for each task τ_i in the given task set.

4.2 Examples

The two following examples display two extremes related to RM scheduling. The first example considers a task set that fully utilizes the processor, yet whose schedule contains idle time. The second example shows that there are cases under which RM produces a valid schedule for task sets whose utilization is equal to one.

4.2.1 Example 1

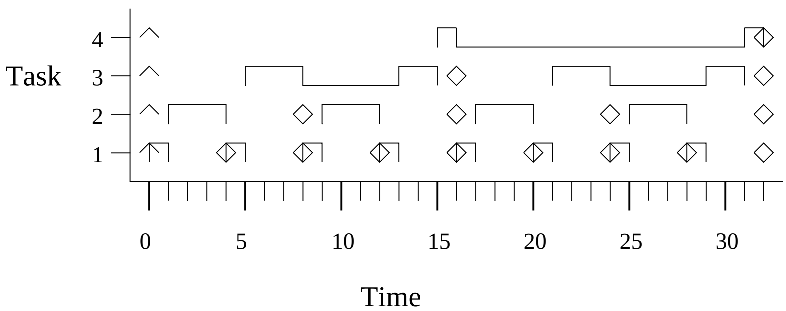
Let T be the task set $\{\tau_i\}_{i=1}^{3}$ such that $\tau_1 = (1, 3, 3, 0)$, $\tau_2 = (1, 4, 4, 0)$, and $\tau_3 = (1, 5, 5, 0)$. Note that by the definition of RM, τ_1 has a higher priority than τ_2, which has a higher priority than τ_3. Hence, at time 0, all three tasks are active, and thus τ_1 executes. Since τ_1 is the highest-priority task, it will not be preempted by any other task. Hence, τ_1 executes to completion at time 1. At time 1, the highest priority active task is τ_2, and it executes until time 2. At time 2, τ_3 is the only active task, so it executes (until time 3). At time 3, τ_1 is released and is the highest priority task (and the only active task). Hence, τ_1 executes until time 4, when τ_2 is the only active task – τ_2 executes until time 5, when τ_3 is released. This process continues, and the RM schedule of T is displayed in the below graph from time 0 to time 30. It is worth noting that if τ_3's execution time were increased, the task set could not be scheduled with RM, because there is only one time unit (from 0 until 5) for τ_3 to execute. Thus, T fully utilizes the processor.



It is interesting to note that the utilization of T is exactly $\frac{1}{3} + \frac{1}{4} + \frac{1}{5} = \frac{47}{60}$. There are 47 of the 60 time units where a task is scheduled, and 13 where the processor is idle.

4.2.2 Example 2

Let T be the task set $\{\tau_i\}_{i=1}^4$ such that $\tau_1 = (1, 4, 4, 0)$, $\tau_2 = (3, 8, 8, 0)$, $\tau_3 = (5, 16, 16, 0)$, and $\tau_4 = (2, 32, 32, 0)$. Note that the utilization of T is exactly 1 – later we will show that RM cannot guarantee schedulability of a task set with n tasks if its utilization is greater than $n(2^{\frac{1}{n}} - 1)$, but that proof does not preclude the possibility that RM can schedule some task sets of utilizations higher than the bound. The graph below displays the RM schedule of T from time 0 to time 32.



Again, we see that the utilization of the task set is represented by the number of time units where a task is scheduled. Namely, the utilization is $\frac{1}{4} + \frac{3}{8} + \frac{5}{16} + \frac{2}{32} = \frac{32}{32}$ (we leave $\frac{32}{32}$ unsimplified to show the relation of the graph to the least common multiple of the task periods). There are 32 time units out of 32 where a task is scheduled, and there is no idle time.

4.3 RM scheduling as an optimal scheduler

We will now show that under certain conditions, RM is optimal among static-priority scheduling algorithms. These conditions are not too demanding, and since RM is easy to understand and simple to implement, it is clear why RM is commonly used in hard real-time scheduling.

4.3.1 Necessary Conditions

For RM, we assume in what follows that deadlines of a given task are concurrent with its releases: for all $i, p_i = d_i$. As well, we assume the system to be synchronous. Since RM is a static-priority scheduling algorithm, (by convention) we assume that tasks are ordered by their priorities, and thus $p_i \leq p_{i+1}$ for all $1 \leq i < n$.

4.3.2 Preliminary Lemmas

We now proceed to prove two lemmas, designed to provide a necessary and sufficient condition for schedule validity under a given static-priority scheduling algorithm. We will then apply that test to RM in order to show that RM is optimal under certain circumstances.

Lemma 4.1 ([LL '73]) *Given a synchronous periodic task set and a fixed-priority scheduling algorithm, a critical instant in the resultant schedule of a given task occurs when that task is requested simultaneously with all higher priority tasks.*

Proof: First, we note that there exists a time when such all tasks are released simultaneously – at time 0 (since the task set is synchronous). Let $t_1, t_2 \in \mathbb{R}_+$ be such that task τ_i has a critical instant at t_1, and completes execution for that release at time t_2. Thus, $t_2 - t_1$ is the longest possible response time for task τ_i.

We claim that each task τ_j with higher-priority than τ_i has exactly $\left\lceil \frac{t_2 - t_1}{p_j} \right\rceil$ releases on $[t_1, t_2)$. Assume, otherwise, that some higher-priority task τ_j has $l < \left\lceil \frac{t_2 - t_1}{p_j} \right\rceil$ releases on $[t_1, t_2)$. Thus, on $[t_1, t_2)$, τ_j executes for a total of $l \cdot e_j$ time units. However, were τ_j released at time t_1, it would be released $\left\lceil \frac{t_2 - t_1}{p_j} \right\rceil$ times on $[t_1, t_2)$. Thus, τ_i's completion at t_2 would be delayed at least $\left(\left\lceil \frac{t_2 - t_1}{p_j} \right\rceil - l \right) e_j$ additional time units, and the satisfaction of τ_i's release at t_1 would be later than time t_2. Again, this implies t_1 is not a critical instant. Additionally, it implies that if every higher priority task τ_j were released at t_1, then each τ_j would have $\left\lceil \frac{t_2 - t_1}{p_j} \right\rceil$ releases on $[t_1, t_2)$.

Hence, a critical instant for a given task occurs when that task is requested simultaneously with all higher priority tasks. $\square$

It is interesting to note that (in the terms of Lemma 4.1) either $t_1 = 0$, or there exists some $\epsilon > 0$ such that on $[t_1 - \epsilon, t_1)$, there is no active task with higher priority than τ_i. Assume that for $t_1 > 0$ and for each $\epsilon > 0$ there is some τ_j active on $[t_1 - \epsilon, t_1)$ with higher priority than τ_i. Then either τ_j or another task with higher priority is scheduled for the interval $[t_1 - \epsilon, t_1)$. Consider that if τ_i were released at time $t_1 - \epsilon$, τ_i would be preempted by higher-priority tasks on $[t_1 - \epsilon, t_1)$. Hence, a release of task τ_i at time $t_1 - \epsilon$ would be satisfied at time t_2, since τ_i would execute on exactly the same intervals as if τ_i were released at t_1. Thus, the release at $t_1 - \epsilon$ would be satisfied at t_2, and τ_i would have a response time of $t_2 - t_1 + \epsilon$. Hence, t_1 would not be a critical instant.

Now that we have a handle on a single task (by way of its critical instant), we procede to a result regarding schedulability of an entire task set.

Lemma 4.2 ([LL '73]) *A static-priority scheduling algorithm produces a valid schedule for a synchronous task set if and only if the first deadline of each task is met.*

Proof: Clearly, if the first deadline of any task is missed, then the task set is not schedulable. Let us then assume that the scheduling algorithm has produced a (possibly valid) schedule for the given task set, and under that schedule, the first deadline of each task is met. From Lemma 4.1, we know that a critical instant occurs when a task is requested simultaneously with all higher priority tasks. Since all tasks are first requested simultaneously, all tasks have a critical instant at that first release. Because all tasks meet their first deadline, the longest response time for any task is exactly the response time for the first release of that task - and since each task meets that first deadline, there is no release of any task that will not be met. □

4.3.3 Proof of optimality

We now proceed to show RM to be an optimal static-priority scheduling algorithm under the condition that for each task in the task set, the task's deadline span is identical to its period. Note that we will actually prove a more general result that will be used in Section 5.

Theorem 4.1 ([LL '73]) *Any static-priority scheduling algorithm where priorities are ordered identically with the task's deadline spans is an optimal scheduling policy among static-priority scheduling algorithms for synchronous task sets.*

Proof: This optimality is shown in [LL '73] via a priority swapping argument. Let T be a task set of n tasks, and since we are considering a static-priority scheduling algorithm, we know that $P_1 < P_2 < \ldots < P_n$. Thus, by the theorem assumption, $d_1 \leq d_2 \leq \ldots \leq d_n$. Now let us assume that there is some valid static-priority schedule g of T. Let the priorities used in g be $P_{g,i}$ for each task τ_i. If $P_{g,i} \leq P_{g,i+1}$ for all $i \in \{1, 2, \ldots n-1\}$, then the priorities are ordered identically with the task deadline spans. Hence, the static-priority scheduling algorithm will produce a valid schedule (as it produces exactly the schedule g).

So, assume there exists $i < j \in \{1, 2, \ldots n\}$ such that $d_i < d_j$ and $P_{g,i} > P_{g,j}$. Without loss of generality, we may assume that there is no τ_k such that $P_{g,i} > P_{g,k} > P_{g,j}$.

Consider the schedule h produced by swapping the priorities of τ_i and τ_j. Formally,

$$\begin{aligned} P_{h,i} &= P_{g,j} \\ P_{h,j} &= P_{g,i} \\ P_{h,k} &= P_{g,k} \qquad \text{for all } k \neq i, j \end{aligned}$$

We will prove that h is also a valid schedule. By swapping the priorities of all such τ_i's and τ_j's, we produce a schedule of the given static-priority scheduling algorithm. Hence, if h is shown to be valid, then the static-priority scheduling algorithm will produce a valid schedule as well.

By Lemma 4.2, if we show that all first deadlines are met by h, then h is valid. Let τ_k be some task of T. Since g is a valid schedule, the first deadline of τ_k is met in g. We can express this result as follows:

$$\sum_{l:P_{g,l} \leq P_{g,k}} \left\lceil \frac{t}{p_l} \right\rceil e_l \leq t \qquad \text{for some } t \leq d_k$$

which states that there is a time $t \leq d_k$ such that τ_k and all higher priority tasks satisfy all of their releases by time t. Additionally, if we substitute h for g and can find such a t, then τ_k will meet its deadline in h.

We now show that all tasks meet their deadlines in h by considering three cases.

Case 1: $\tau_k \neq \tau_i$ and $\tau_k \neq \tau_j$. See then that $\{l : P_{g,l} \leq P_{g,k}\} = \{l : P_{h,l} \leq P_{h,k}\}$. Since τ_k meets its deadline in g, there exists a $t \leq d_k$ such that

$$\sum_{l:P_{g,l} \leq P_{g,k}} \left\lceil \frac{t}{p_l} \right\rceil e_l \leq t$$

by substitution, we have

$$\sum_{l:P_{h,l} \leq P_{h,k}} \left\lceil \frac{t}{p_l} \right\rceil e_l \leq t \leq d_k$$

Thus, τ_k meets its deadline in h.

Case 2: $\tau_k = \tau_j$. Since $P_{g,i} = P_{h,j}$, $\{l : P_{g,l} \leq P_{g,i}\} = \{l : P_{h,l} \leq P_{h,j}\}$. Since τ_i meets its deadline in g, there is some $t_i \leq d_i$ such that

$$\sum_{l:P_{g,l} \leq P_{g,i}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l \leq t_i$$

by substitution, we have

$$\sum_{l:P_{h,l} \leq P_{h,j}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l \leq t_i \leq d_i \leq d_j$$

Thus, τ_j meets its deadline in h.

Prior to examining the other case, we first note (as in Case 2) that there exists $t_i \leq d_i$ such that

$$\sum_{l:P_{g,l} \leq P_{g,i}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l \leq t_i$$

Recalling that there is no τ_k with $P_{g,j} < P_{g,k} < P_{g,i}$, we separate the sum as follows:

$$\left(\sum_{l:P_{g,l} \leq P_{g,j}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l \right) + \left(\sum_{l:P_{g,l} = P_{g,i}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l \right) \leq t_i$$

By assumption for static-priority algorithms, there are no identical priorities:

$$\left(\sum_{l:P_{g,l} \leq P_{g,j}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l \right) + \left\lceil \frac{t_i}{p_i} \right\rceil e_i \leq t_i \tag{1}$$

Case 3: $\tau_k = \tau_i$. Let t_i be as described in Case 2. Then we have

$$\sum_{l:P_{h,l} \leq P_{h,i}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l = \left(\sum_{l:P_{g,l} \leq P_{g,j}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l \right) - \left\lceil \frac{t_i}{p_j} \right\rceil e_j + \left\lceil \frac{t_i}{p_i} \right\rceil e_i$$

by equation (1),

$$\begin{aligned} \sum_{l:P_{h,l} \leq P_{h,i}} \left\lceil \frac{t_i}{p_l} \right\rceil e_l &\leq t_i - \left\lceil \frac{t_i}{p_j} \right\rceil e_j \\ &\leq t_i \leq d_i \end{aligned}$$

Hence, τ_i meets its deadline in h.

Thus, for any $\tau_k \in T$, if τ_k meets its first deadline in g, then τ_k meets its first deadline in h. Therefore, if g is a valid schedule of T, then so is h. $\square$

Corollary *RM is an optimal scheduling algorithm among static-priority scheduling algorithms for synchronous task sets where each task's deadline span and period are identical.*

This corollary follows directly from Theorem 4.1 since under RM, priorities are ordered by period lengths. Since we are assuming period lengths are equal to deadline spans, then the theorem holds for RM under the given restrictions.

4.4 Utilization Results

One of the more powerful results of [LL '73] is the derivation of a sufficient test of schedulability under RM related to the utilization of the given task set. This development hinges on the determination of a "worst case" task set – one that minimizes utilization while fully utilizing the processor. In this section, we will re-develop some of [LL '73]'s results based upon several lemmas that we create from the works of [LL '73] and [LSD '89].

We now proceed to develop mathematical tests for schedulability and full utilization. With these lemmas in hand, we will be able to answer the "worst case" task set question, and define the task sets that minimize utilization while fully utilizing the processor.

For schedulability, by Lemma 4.2, we only need to concern ourselves with a task's first deadline. The next Lemma provides an equation to determine if a given task's first deadline is met.

Lemma 4.3 *Let T be a synchronous task set and g be a static priority schedule of T such that $P_1 < P_2 < \ldots < P_n$. Given $\tau_i \in T$, there exists a $t \leq p_i$ such that $\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq t$ if and only if τ_i satisfies its first release at or before time p_i.*

Proof: We first assume that there exists a $t \leq p_i$ such that $\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq t$. We will show that by time t, task τ_i satisfies its first release at or before time t. Assume otherwise, that τ_i is still active at time t. Then $\int_0^t \chi_{g,\tau_i}(x)dx < e_j$. Let $w = \int_0^t \chi_{g,\tau_i}(x)dx$. The total amount of work requested by tasks with higher priority than τ_i on $[0,t)$ is $\sum_{j=1}^{i-1}\left\lceil \frac{t}{p_j}\right\rceil e_j$. The maximum amount of work due to τ_i and higher priority tasks on $[0,t)$ is then $w+\sum_{j=1}^{i-1}\left\lceil \frac{t}{p_j}\right\rceil e_j < \sum_{j=1}^{i}\left\lceil \frac{t}{p_j}\right\rceil e_j \le t$. Therefore, there is some time $t_0 \in [0,t)$ such that at time t_0, the processor is idle, or a lower priority (than that of τ_i) task is scheduled. Note, though, that at t_0, τ_i is active since $t_0 < t$ and τ_i is active at time t. By the definition of a static priority scheduling algorithm, τ_i (or a higher priority task) must be scheduled at time t_0. By contradiction, τ_i is not active at time t, and therefore has satisfied its release at time 0.

We now assume that there exists a $t_0 \le p_i$ such that τ_i satisfies its first release at or before time t_0. We must show that there exists a $t \le p_i$ such that $\sum_{j=1}^{i}\left\lceil \frac{t}{p_j}\right\rceil e_j \le t$. Since τ_i has satisfied its first relase by time t_0, there is some time $t \le t_0$ such that τ_i has satisfied its first release by time t, and for any $\epsilon > 0$, τ_i is still active at time $t-\epsilon$. Therefore, we know that there exists a $t_1 \le t$ such that τ_i is scheduled in g on the interval $[t_1,t)$. Thus, by definition of a static priority scheduling algorithm, τ_i is the highest priority active task at any time on $[t_1,t)$. Thus, at time t, we know that tasks $\tau_1,\tau_2,\ldots,\tau_{i-1}$ have satisfied all their releases prior to time t. Therefore, every release of tasks $\tau_1,\tau_2,\ldots,\tau_i$ on $[0,t)$ is satisfied at or before time t. Hence, $\sum_{j=1}^{i}\left\lceil \frac{t}{p_j}\right\rceil e_j \le t$, and $t \le p_i$. □

Having shown how to determine if a particular task meets its first deadline, we now apply that knowledge to the whole task set to create a necessary and sufficient test of schedulability.

Lemma 4.4 ([LSD '89]) *Let T be a synchronous task set and g be a static priority schedule of T such that $P_1 < P_2 < \ldots < P_n$. g is a valid schedule of T if and only if*

$$\max_{\tau_i \in T}\left\{ \min_{t \in \left\{ k\cdot p_j \mid j \le i, k \in \left\{1,\ldots,\left\lfloor \frac{p_i}{p_j}\right\rfloor\right\}\right\}} \left\{ \frac{1}{t}\sum_{j=1}^{i}\left\lceil \frac{t}{p_j}\right\rceil e_j \right\}\right\} \le 1 \qquad (2)$$

Proof: By Lemma 4.3, we know that a given task τ_i meets its first deadline if and only if there exists a $t \le p_i$ such that $\sum_{j=1}^{i}\left\lceil \frac{t}{p_j}\right\rceil e_j \le t$. By Lemma 4.2, we know that every task meets its first deadline if and only if the schedule is valid. Therefore we have the following chain of equivalent statements:

The schedule is valid if and only if for each $\tau_i \in T$, there exists a $t \le p_i$ such that $\sum_{j=1}^{i}\left\lceil \frac{t}{p_j}\right\rceil e_j \le t$.

The schedule is valid if and only if for each $\tau_i \in T$, there exists a $t \le p_i$ such that $\frac{1}{t}\sum_{j=1}^{i}\left\lceil \frac{t}{p_j}\right\rceil e_j \le 1$.

The schedule is valid if and only if for all $\tau_i \in T$,

$$\min_{t\in[0,p_i]} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} \leq 1.$$

The schedule is valid if and only if

$$\max_{\tau_i \in T} \left\{ \min_{t\in[0,p_i]} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} \right\} \leq 1. \tag{3}$$

Note that we may restrict our consideration for the value of t from the set $[0, p_i]$ to the set $S = \left\{ k \cdot p_j | j \leq i, k \in \{1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\}$. The set S represents every deadline on $[0, p_i]$ of all tasks τ_j with $P_j \leq P_i$. We claim that $\frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j$ achieves its minimums on the set S. That is to say, we claim that for all $t_1 \notin S$, there exists a $t_2 \in S$ such that $\frac{1}{t_1} \sum_{j=1}^{i} \left\lceil \frac{t_1}{p_j} \right\rceil > \frac{1}{t_2} \sum_{j=1}^{i} \left\lceil \frac{t_2}{p_j} \right\rceil$. Let $t_1 \in [0, p_i)$ such that $t_1 \notin S$. Since $t_1 < p_i$ and $p_i \in S$, there is a $t_2 \in S$ such that $t_1 < t_2$. Let t_2 be the minimal element in S that is greater than t_1. Thus, for $1 \leq j \leq i$, $\left\lceil \frac{t_1}{p_j} \right\rceil = \left\lceil \frac{t_2}{p_j} \right\rceil$. To see this claim, assume otherwise, that there exists a $j \in \{1, \ldots, i\}$ such that $\left\lceil \frac{t_1}{p_j} \right\rceil \neq \left\lceil \frac{t_2}{p_j} \right\rceil$. Since $t_1 < t_2$, $\left\lceil \frac{t_1}{p_j} \right\rceil < \left\lceil \frac{t_2}{p_j} \right\rceil$. Let $k = \left\lceil \frac{t_1}{p_j} \right\rceil$. We know $t_1 \neq kp_j$ since $t_1 \notin S$. Also, $kp_j \in S$ by the definition of S. Therefore, $t_1 < kp_j < t_2$. However, we defined t_2 to be the minimal element of S that is greater than t_1. By contradiction, we have shown that for $1 \leq j \leq i$, $\left\lceil \frac{t_1}{p_j} \right\rceil = \left\lceil \frac{t_2}{p_j} \right\rceil$. Thus,

$$\begin{aligned} \frac{1}{t_1} \sum_{j=1}^{i} \left\lceil \frac{t_1}{p_j} \right\rceil e_j &= \frac{1}{t_1} \sum_{j=1}^{i} \left\lceil \frac{t_2}{p_j} \right\rceil e_j \\ &> \frac{1}{t_2} \sum_{j=1}^{i} \left\lceil \frac{t_2}{p_j} \right\rceil e_j \end{aligned}$$

Therefore, for any $t_1 \notin S$ there is a $t_2 \in S$ such that $\frac{1}{t_2} \sum_{j=1}^{i} \left\lceil \frac{t_2}{p_j} \right\rceil e_j < \frac{1}{t_1} \sum_{j=1}^{i} \left\lceil \frac{t_1}{p_j} \right\rceil e_j$. Hence, $\frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j$ will acheive its minimum on S.

Therefore, we have

$$\min_{t\in[0,p_i]} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = \min_{t \in \left\{ k\cdot p_j | j \leq i, k \in \{1,\ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\}} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} \tag{4}$$

Combining equations (3) and (4), we have the desired result: g is a valid schedule of T if and only if

$$\max_{\tau_i \in T} \left\{ \min_{t \in \left\{ k \cdot p_j | j \leq i, k \in \{1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\}} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} \right\} \leq 1$$

□

Thus, our first goal of this section has been met: We have a computable test for schedulability. We now build upon that knowledge for a test of full utilization. The first step in that process is to show that full utilization is based upon idle time prior to the last first deadline.

Lemma 4.5 ([LL '73]) *A synchronous task set fully utilizes the processor under a static-priority scheduling algorithm if and only if there is no idle time prior to time p_n and all deadlines at or prior to p_n are satisfied.*

Proof: By Lemma 4.2, all deadlines prior to p_n must be satisfied for the task set to be schedulable. By definition, a task set that fully utilizes the processor must be schedulable. Thus, we focus our consideration on idle time prior to p_n.

If there is idle time prior to p_n, then the execution time of the task with the longest period may be increased by the amount of idle time. Since all first deadlines are still met, the static-priority scheduling algorithm (hereafter denoted SPSA) does yield a valid schedule for the modified task set. Since a task's execution could be increased and still the task set would be schedulable, the original task set did not fully utilize the processor. Thus, if a task set fully utilizes the processor under SPSA, there is no idle time prior to $\max\{p_i\}$.

If there is no idle time prior to p_n and there is some $\tau_j \in T$ and $\epsilon > 0$ such that replacing e_j by $e_j + \epsilon$ yields a valid schedule under SPSA, consider task τ_n. In the original schedule, let w be the amount of time the processor is devoted to tasks $\tau_1, \tau_2, \ldots, \tau_{n-1}$ on $[0, p_n)$. Thus, since there is no idle time prior to p_n, $w + e_n = p_n$. However, in the modified task set, τ_n will miss its deadline at p_n: Consider that in the SPSA schedule of the modified task set, if $j < n$, then the amount of time necessary for the tasks $\tau_1, \tau_2, \ldots, \tau_{n-1}$ on the interval $[0, p_n)$ will be at least $w + \epsilon$, since τ_j has at least one deadline prior to p_n. Since τ_n may only execute when other tasks are inactive (because τ_n is the lowest priority task), in the modified task set schedule, τ_n has $p_n - w - \epsilon$ time units to execute. Since $p_n - w - \epsilon < e_n$, τ_n will miss its deadline at p_n. If $j = n$, then consider that tasks $\tau_1, \tau_2, \ldots, \tau_{n-1}$ occupy w time units in $(0, p_n)$ in the modified task set schedule, leaving e_n time units for τ_n to complete $e_n + \epsilon$

time units of computation. Thus, there is no execution time that may be increased in the original task set without yielding an invalid schedule under SPSA, and the original task set fully utilizes the processor under SPSA. □

Since we have seen full utilization is based on idle time prior to the last first deadline, we now consider idle time prior to a given task's first deadline. We use Lemma 4.4 to determine if a static priority schedule of a synchronous task set contains idle time prior to a given task's first deadline.

Lemma 4.6 *Let T be a synchronous task set and g be a valid static priority schedule of T such that $P_1 < P_2 < \ldots < P_n$. For any task $\tau_i \in T$, there is no idle time in g on the interval $[0, p_i)$ if and only if*

$$\min_{t \in \left\{ k \cdot p_j | j \leq i, k \in \{1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\}} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1. \tag{5}$$

Proof: Let $\tau_i \in T$. For ease of notation, we define the set $S = \left\{ k \cdot p_j | j \leq i, k \in \{1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\}$.

Assume there is no idle time on $[0, p_i)$. Since the schedule is valid, then by Lemma 4.4, $\min_{t \in S} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} \leq 1$. By contradiction, we will show that equation (5) holds. Assume that there exists a $t \in S$ such that $\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j < t$. Then there exists some $t_0 < t$ such that $\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j = t_0$. Since there is no idle time on $[0, t_0)$, all task requests on $[0, t)$ must be satisfied by time t_0. Hence, there is no active task on $[t_0, t)$ – the processor is then idle, contradicting our assumption that there is no idle time on $[0, p_i)$. Thus, $\min_{t \in S} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1$.

Assume that equation (5) holds: $\min_{t \in S} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1$. We must show that there is no idle time on $[0, p_i)$. Again, we do this by contradiction. Assume there is idle time $[t_0, t)$ on $[0, p_i)$. Since there is an active task when a release occurs, we may assume that a release occurs at time t, denoting the end of the idle period. Therefore, t is in the set S. Since equation (5) holds, $\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \geq t$. However, if the processor is idle on $[t_0, t)$, then we know all task requests at or before t are satisfied by time t_0. Thus, $\sum_{j=1}^{i} \left\lceil \frac{t_0}{p_j} \right\rceil e_j \leq t_0$. Since $t_0 < t$, we have a contradiction, and there can be no such idle time $[t_0, t)$.

We have shown that given a valid schedule, there is no idle time on $[0, p_i)$ if and only if

$\min_{t \in S} \left\{ \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = t$. □

Having a means of testing full utilization in terms of idle time and first deadlines, and a computational means to determine idle time, we have the tools we need to proceed. We combine the previous two results to provide a computational test to determine if a task set fully utilizes the processor under a given schedule.

Lemma 4.7 *Let T be a synchronous task set and g be a static priority schedule of T such that $P_1 < P_2 < \ldots < P_n$. T fully utilizes the processor under g if and only if for all $\tau_i \in T, 1 \leq i < n$*

$$\textit{there exists a } t \in \left\{ k \cdot p_j | j \leq i, k \in \{1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\} \textit{ such that } \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq t \tag{6}$$

and

$$\min_{t \in \left\{ k \cdot p_j | j \leq n, k \in \{1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\}} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1. \tag{7}$$

Proof: By Lemma 4.5, we know that a task set fully utilizes a processor if and only if the the schedule is valid, and there is no idle time prior to time p_n. We have seen in Lemma 4.4 that a task set's schedule is valid if and only if equation (2) holds. By Lemma 4.6, we know that given a valid schedule, there is no idle time prior to p_n if and only if equation (7) holds.

We first assume that equations (6) and (7) hold. Clearly, these two equations imply equation (2). As well, equation (7) implies there is no idle time prior to time p_n. Thus, by Lemma 4.5, we know that equations (6) and (7) imply the task set fully utilizes the processor under the given schedule.

Assuming the task set fully utilizes the processor, then the schedule is valid and there is no idle time prior to time p_n. Thus, equation (2) holds, and there is no idle time prior to p_n. Equation (2) implies equation (6), and the lack of idle time implies equation (7). Thus, if a task set fully utilizes the processor, equations (6) and (7) hold. □

We will work extensively with Lemma 4.7 in the following proof, and before we begin the proof proper, we will make use of some assumptions on period length to considerably simplify the set over which t is considered in equations (6) and (7).

Lemma 4.8 *For $\{p_i\}_{i=1}^n$ with $p_1 \le p_2 \le \ldots \le p_n$ and $\frac{p_n}{p_1} \le 2$,*

$$\left\{k \cdot p_j | j \le i, k \in \left\{1, 2, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \right\}\right\} = \{p_j\}_{j=1}^i$$

Proof: Since $p_1 \le p_2 \le \ldots \le p_n$ and $\frac{p_n}{p_1} \le 2$, we know that for any $\tau_i \in T, j < i$, we have $p_j \le p_i$ and $2p_j \ge p_i$. Thus, $\left\lfloor \frac{p_i}{p_j} \right\rfloor \le 2$ with equality if and only if $2p_j = p_i$. The remainder of the proof should be clear. □

[LL '73] states the following result, but their proof is faulty (as will be shown below). The theorem lays the groundwork for providing a necessary condition for schedulability under RM. The significance of the condition is that it may be tested in linear time, whereas a necessary and sufficient test of schedulability requires psuedo-polynomial time (as will be seen in Section 4.6).

Theorem 4.2 ([LL '73]) *Over the set of synchronous task sets with n tasks that fully utilize the processor under RM such that $p_1 \le p_2 \le \cdots \le p_n$ and $\frac{p_n}{p_1} \le 2$, the execution times $e_i = p_{i+1} - p_i$ for $1 \le i < n$ and $e_n = 2p_1 - p_n$ minimize utilization.*

We prove this Theorem by subdividing into three cases based on the execution times of tasks τ_1 through τ_{n-1}. In cases 1 and 2, we will modify the task set in such a way that the modified task set fully utilizes the processor and whose utilization is less than or equal to that of the original task set. Repeated modifications will convert the task set into one where the execution times for tasks τ_1 through τ_{n-1} are identical to the times listed in the statement of the Theorem. Case 3 will show that given such execution times for τ_1 through τ_{n-1}, task τ_n must have the execution time specified above.

Throughout this theorem, we will make use of Lemma 4.8. Since the task set satisfies the conditions of that lemma, the set over which we must consider t in equations (6) and (7) is merely $\{p_j\}_{j=1}^i$.

Proof: Since T fully utilizes the processor, we know that it satisfies the two conditions of Lemma 4.7, namely equation (6): for all $\tau_k \in T$,

$$\text{there exists a } t \in \{p_j\}_{j=1}^k \text{ such that } \sum_{j=1}^k \left\lceil \frac{t}{p_j} \right\rceil e_j \le t$$

and equation (7):

$$\min_{t \in \{p_j\}_{j=1}^{n}} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1.$$

We aim to prove the same for T', defined below.

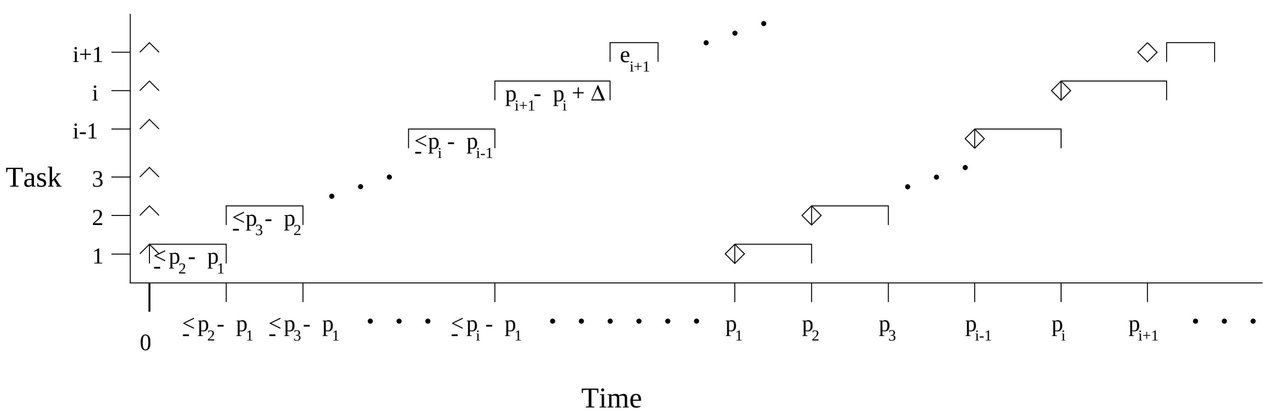
Case 1: There exists an $i < n$ such that $e_i > p_{i+1} - p_i$.

Let $i < n$ be the lowest indexed such e_i. Then let $\Delta = e_i - (p_{i+1} - p_i)$. Hence, $e_i = p_{i+1} - p_i + \Delta$. Consider the task set T', identical to T except for execution times:

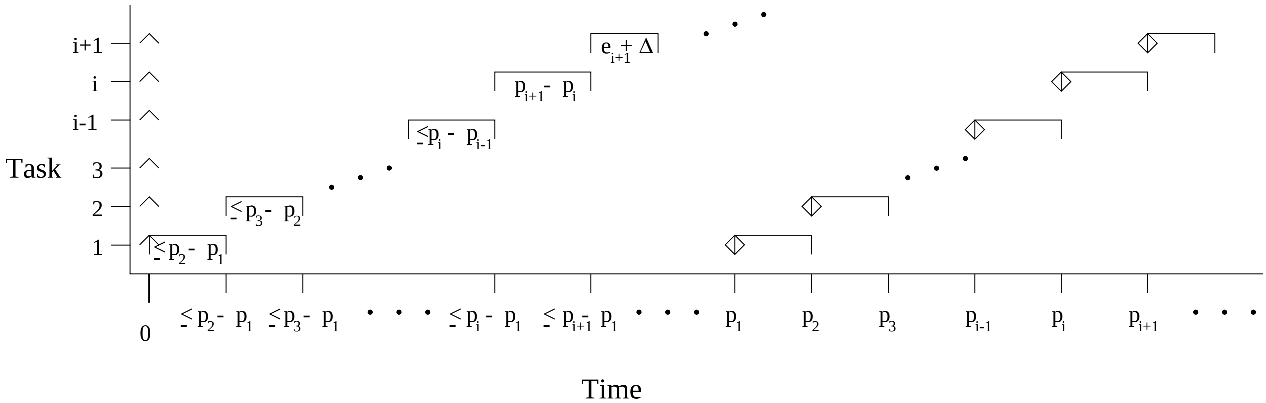
$$\begin{aligned} e'_i &= p_{i+1} - p_i \\ e'_{i+1} &= e_{i+1} + \Delta \\ e'_j &= e_j \quad \text{for all} \quad j \neq i, i+1 \end{aligned}$$

Let g be the schedule produced by RM for T, and g' be the schedule produced by RM for T'.

Here is an example of how those schedules appear. Note the change that occurs immediately after time p_{i+1}.



Case 1 sample graph of T with period lengths indicated



Case 1 corresponding graph of T' with period lengths indicated

Case 1: Subproof that T' fully utilizes the processor: We must satisfy the two conditions of Lemma 4.7, equations (6) and (7) above.

First, we handle equation (6). Let $\tau_k \in T$. Since T fully utilizes the processor, by equation (6) we have

$$\text{there exists a } t \in \{p_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq t$$

Since $p_j = p'_j$ for all $1 \leq j \leq n$,

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e_j \leq t \tag{8}$$

Now we divide our consideration into three subcases based on the value of k in relation to i. Our goal in each case is to show that there exists a $t \in \{p'_j\}_{j=1}^{k}$ such that

$$\sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t,$$

thereby proving T' satisfies equation (6).

Subcase 1.A: $k < i$. In this case, $e_j = e'_j$ for all $1 \leq j \leq k$.

Therefore, equation (8) becomes

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

Thus, equation (6) holds for subcase 1.A.

Subcase 1.B: $k = i$.

We now break up the sum from equation (8) to produce the desired result:

$$\begin{aligned}
\sum_{j=1}^{i} \left\lceil \frac{t}{p'_j} \right\rceil e_j &= \left\lceil \frac{t}{p'_i} \right\rceil e_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{p'_j} \right\rceil e_j \\
&= \left\lceil \frac{t}{p'_i} \right\rceil (p_{i+1} - p_i + \Delta) + \sum_{j=1}^{i-1} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \\
&= \left\lceil \frac{t}{p'_i} \right\rceil (p_{i+1} - p_i) + \left\lceil \frac{t}{p'_i} \right\rceil \Delta + \sum_{j=1}^{i-1} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \\
&= \left\lceil \frac{t}{p'_i} \right\rceil e'_i + + \sum_{j=1}^{i-1} \left\lceil \frac{t}{p'_j} \right\rceil e'_j + \left\lceil \frac{t}{p'_i} \right\rceil \Delta \\
&= \sum_{j=1}^{i} \left\lceil \frac{t}{p'_j} \right\rceil e'_j + \left\lceil \frac{t}{p'_i} \right\rceil \Delta
\end{aligned}$$

Since $0 < p'_j \le p'_i$ for all $t \in \{p'_j\}_{j=1}^{i}$, $\left\lceil \frac{t}{p'_i} \right\rceil = 1$ and we have

$$\sum_{j=1}^{i} \left\lceil \frac{t}{p'_j} \right\rceil e_j = \sum_{j=1}^{i} \left\lceil \frac{t}{p'_j} \right\rceil e'_j + \Delta$$

Therefore, by equation (8), we have

$$\begin{aligned}
\text{there exists a } t \in \{p'_j\}_{j=1}^{i} \text{ such that } \sum_{j=1}^{i} \left\lceil \frac{t}{p'_j} \right\rceil e'_j + \Delta &\le t \\
\text{there exists a } t \in \{p'_j\}_{j=1}^{i} \text{ such that } \sum_{j=1}^{i} \left\lceil \frac{t}{p'_j} \right\rceil e'_j &< t
\end{aligned}$$

Thus, equation (6) holds for subcase 1.B.

Subcase 1.C: $k > i$.

As in subcase 1.B, we will break down equation (8)'s sum for analysis. Again, we use the fact that for all $j \in \{1, 2, \ldots, n\}$, $j \notin \{i, i+1\}$, we have $e'_j = e_j$.

$$\sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e_j = \sum_{j=1}^{i-1} \left\lceil \frac{t}{p'_j} \right\rceil e'_j + \left\lceil \frac{t}{p'_i} \right\rceil e_i + \left\lceil \frac{t}{p'_{i+1}} \right\rceil e_{i+1} + \sum_{j=i+2}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j$$

$$\begin{aligned}
&= \sum_{j=1}^{i-1}\left\lceil\frac{t}{p'_j}\right\rceil e'_j + \left\lceil\frac{t}{p'_i}\right\rceil (p_{i+1}-p_i+\Delta) + \left\lceil\frac{t}{p'_{i+1}}\right\rceil (e'_{i+1}-\Delta) + \sum_{j=i+2}^{k}\left\lceil\frac{t}{p'_j}\right\rceil e'_j \\
&= \sum_{j=1}^{i-1}\left\lceil\frac{t}{p'_j}\right\rceil e'_j + \left\lceil\frac{t}{p'_i}\right\rceil (p_{i+1}-p_i) + \left\lceil\frac{t}{p'_i}\right\rceil \Delta + \left\lceil\frac{t}{p'_{i+1}}\right\rceil e'_{i+1} - \left\lceil\frac{t}{p'_{i+1}}\right\rceil \Delta \\
&\qquad + \sum_{j=i+2}^{k}\left\lceil\frac{t}{p'_j}\right\rceil e'_j \\
&= \sum_{j=1}^{i-1}\left\lceil\frac{t}{p'_j}\right\rceil e'_j + \left\lceil\frac{t}{p'_i}\right\rceil e'_i + \left\lceil\frac{t}{p'_i}\right\rceil \Delta - \left\lceil\frac{t}{p'_{i+1}}\right\rceil \Delta + \sum_{j=i+1}^{k}\left\lceil\frac{t}{p'_j}\right\rceil e'_j \\
&= \sum_{j=1}^{k}\left\lceil\frac{t}{p'_j}\right\rceil e'_j + \left(\left\lceil\frac{t}{p'_i}\right\rceil - \left\lceil\frac{t}{p'_{i+1}}\right\rceil\right)\Delta
\end{aligned}$$

Therefore, if $\left\lceil\frac{t}{p'_i}\right\rceil = \left\lceil\frac{t}{p'_{i+1}}\right\rceil$,

$$\sum_{j=1}^{k}\left\lceil\frac{t}{p'_j}\right\rceil e_j = \sum_{j=1}^{k}\left\lceil\frac{t}{p'_j}\right\rceil e'_j$$

which would yield our desired result for this subcase. So we now focus our attention on the set of values of t, namely $\{p_j\}_{j=1}^{k}$. For $t = p'_j \leq p_i$ (therefore $j \leq i$), we know $t \leq p_{i+1}$, so $\left\lceil\frac{t}{p_i}\right\rceil = 1$ and $\left\lceil\frac{t}{p_{i+1}}\right\rceil = 1$. For $t = p'_j > p_{i+1}$, we know $p_i \leq p_{i+1} < t < 2p_i \leq 2p_{i+1}$, so $\left\lceil\frac{t}{p_i}\right\rceil = 2$ and $\left\lceil\frac{t}{p_{i+1}}\right\rceil = 2$. Therefore, the only value of $t \in \{p_j\}_{j=1}^{k}$ where $\left\lceil\frac{t}{p'_i}\right\rceil \neq \left\lceil\frac{t}{p'_{i+1}}\right\rceil$ is $t = p_{i+1}$ (when $p_i \neq p_{i+1}$). Thus, my goal is to show that when $p_i \neq p_{i+1}$, equation (8) holds true for some value of t other than p_{i+1}. To do so, we must show that

$$\sum_{j=1}^{k}\left\lceil\frac{p_{i+1}}{p'_j}\right\rceil e_j > p_{i+1}$$

First, we note that since $p_i < p_{i+1}$, for all $1 \leq j \leq i$, $\left\lceil\frac{p_{i+1}}{p_j}\right\rceil = 2$ and for all $i+1 \leq j \leq k$, $\left\lceil\frac{p_{i+1}}{p_j}\right\rceil = 1$. Now, we analyze the sum

$$\begin{aligned}
\sum_{j=1}^{k}\left\lceil\frac{p_{i+1}}{p'_j}\right\rceil e_j &= \sum_{j=1}^{i} 2e_j + \sum_{j=i+1}^{k} 1e_j \\
&= \sum_{j=1}^{i-1} 2e_j + 1e_i + \sum_{j=i+1}^{k} 1e_j + 1e_i \\
&= \sum_{j=1}^{i-1} 2e_j + \sum_{j=i}^{k} 1e_j + e_i \qquad (9)
\end{aligned}$$

Now, by the case 1 assumption, τ_i is the lowest indexed task such that $e_i > p_{i+1} - p_i$. Therefore, $e_{i-1} \leq p_i - p_{i-1}$. Thus, since $e_{i-1} > 0$, we know $p_{i-1} < p_i$. We have $\left\lceil \frac{p_i}{p'_j} \right\rceil = 2$ for all $j < i$, $\left\lceil \frac{p_i}{p'_j} \right\rceil = 1$ for all $j \geq i$. Combining that knowledge with equation (9), we get

$$\begin{aligned} \sum_{j=1}^{k} \left\lceil \frac{p_{i+1}}{p'_j} \right\rceil e_j &= \sum_{j=1}^{i-1} \left\lceil \frac{p_i}{p'_j} \right\rceil e_j + \sum_{j=i}^{k} \left\lceil \frac{p_i}{p'_j} \right\rceil e_j + e_i \\ &= \sum_{j=1}^{k} \left\lceil \frac{p_i}{p'_j} \right\rceil e_j + e_i \qquad (10) \end{aligned}$$

Now, since T fully utilizes the processor, then there is no idle time prior to time p_n. More importantly here, there is no idle time prior to time p_i. Therefore, by equation (5), we know

$$\sum_{j=1}^{i} \left\lceil \frac{p_i}{p'_j} \right\rceil e_j \geq p_i$$

Since $k > i$,

$$\sum_{j=1}^{k} \left\lceil \frac{p_i}{p'_j} \right\rceil e_j > p_i$$

Combined with equation (10), we then have

$$\begin{aligned} \sum_{j=1}^{k} \left\lceil \frac{p_{i+1}}{p'_j} \right\rceil e_j &> p_i + e_i \\ &= p_i + (p_{i+1} - p_i + \Delta) \\ &= p_{i+1} + \Delta \end{aligned}$$

Therefore, we know that when $p_i \neq p_{i+1}$, for $t = p_{i+1}$, $\sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e_j > t$. Since equation (8) holds true, then it holds true for some $t \neq p_{i+1}$. We then have

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k}, t \neq p_{i+1} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e_j \leq t$$

Thus, t is such that $\left\lceil \frac{t}{p'_i} \right\rceil = \left\lceil \frac{t}{p'_{i+1}} \right\rceil$, and

$$\sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e_j = \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \qquad (11)$$

yielding

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k}, t \neq p_{i+1} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

Therefore,

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k}, \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

and equation (6) holds for subcase 1.C.

Having considered all subcases, we have shown that in case 1, the task set T' satisfies equation (6).

We now prove that for case 1, T' satisfies equation (7). First note that $k = n$ falls into subcase 1.C above since $k > i$ for all $i < n$, and case 1 assumes $i < n$. With that knowledge, for $k = n$ we have the following by equation (11):

$$\sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j = \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j$$

Since T fully utilizes the processor, equation (7) holds:

$$\min_{t \in \{p_j\}_{j=1}^{n}} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1.$$

Combining the previous two equations with the knowledge that $p_j = p'_j$ for all $1 \leq j \leq n$, we have

$$\min_{t \in \{p'_j\}_{j=1}^{n}} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \right\} = 1.$$

Thus, in case 1, T' satisfies equation (7).

Since T' satisfies both requirements of Lemma 4.7, then we have shown that in case 1, T' fully utilizes the processor.

Case 1: Subproof that the utilization of T' is at most that of T: The utilization of T is

$$U = \frac{e_1}{p_1} + \frac{e_2}{p_2} + \cdots + \frac{e_{i-1}}{p_{i-1}} + \frac{p_{i+1} - p_i + \Delta}{p_i} + \frac{e_{i+1}}{p_{i+1}} + \cdots + \frac{e_n}{p_n}$$

The utilization of T' is

$$U' = \frac{e_1}{p_1} + \frac{e_2}{p_2} + \cdots + \frac{e_{i-1}}{p_{i-1}} + \frac{p_{i+1} - p_i}{p_i} + \frac{e_{i+1} + \Delta}{p_{i+1}} + \cdots + \frac{e_n}{p_n}$$

Hence, the difference in utilization is

$$\begin{aligned} U - U' &= \frac{\Delta}{p_i} - \frac{\Delta}{p_{i+1}} \\ &= \frac{p_{i+1}\Delta - p_i\Delta}{p_i p_{i+1}} \\ &= \frac{p_{i+1} - p_i}{p_i p_{i+1}}\Delta \end{aligned}$$

since Δ, p_i, and p_{i+1} are positive, and $p_{i+1} \geq p_i$, we have

$$U - U' = \frac{p_{i+1} - p_i}{p_i p_{i+1}}\Delta \geq 0$$

with equality if and only if $p_i = p_{i+1}$. Thus, the utilization of T' is less than or equal to the utilization of T.

Thus, for case 1, we have provided a task set with a utilization at most that of T that fully utilizes the processor. Additionally, we know that for T', there is one less task (than in T) τ'_i such that $e'_i > p'_{i+1} - p'_i$. Since there are a finite number of tasks in the task set, we may apply the case 1 transformation repeatedly, until we know that for all $i < n$, $e_i \leq p_{i+1} - p_i$. Specifically, repeated transformations will eventually yield a task set whose utilization is at most that of the original task set, that fully utilizes the processor, and which falls into case 2 or 3 below.

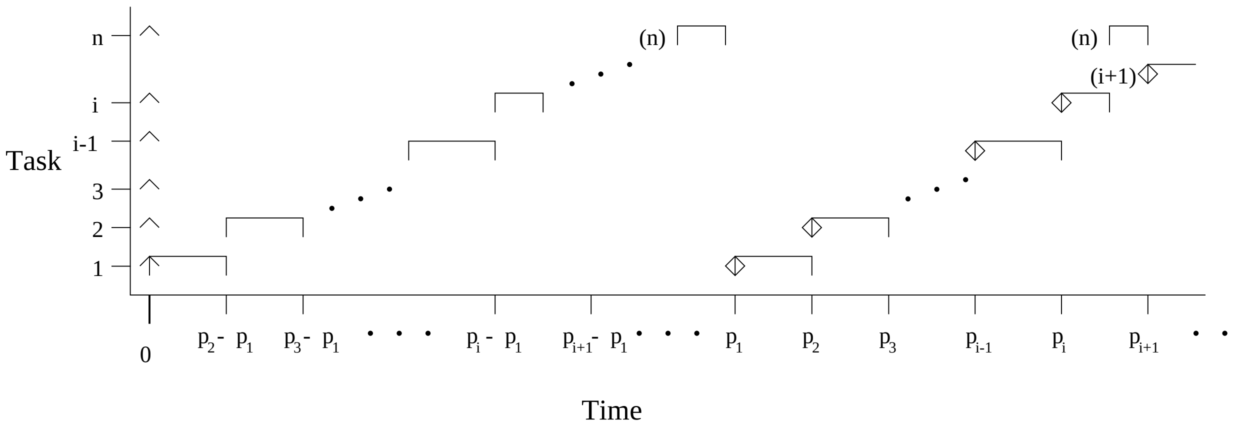
Case 2: For all $i < n$, $e_i \leq p_{i+1} - p_i$ and there is some e_i such that $e_i < p_{i+1} - p_i$.

Let $i < n$ be the lowest indexed such e_i. Then let $\Delta = (p_{i+1} - p_i) - e_i$. Hence, $e_i = p_{i+1} - p_i - \Delta$. Consider the task set T', identical to T except for execution times:

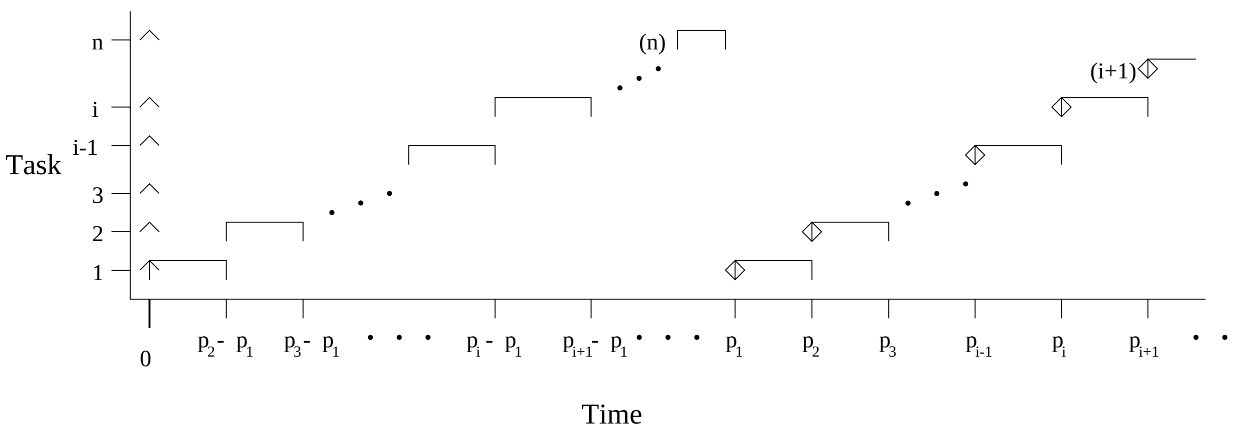
$$\begin{aligned} e'_i &= p_{i+1} - p_i \\ e'_n &= e_n - 2\Delta \\ e'_j &= e_j \quad \text{for all} \quad j \neq i, i+1 \end{aligned}$$

Let g be the schedule produced by RM for the T, and g' be the schedule produced by RM for T'.

Here is an example of how those schedules appear. Note the change that occurs immediately before times $p_{i+1} - p_1$ and p_{i+1}.



Case 2 sample graph of T



Case 2 corresponding graph of T'

Case 2: Subproof that T' fully utilizes the processor: We must satisfy the two conditions of Lemma 4.7, equation (6): for all $\tau'_k \in T'$,

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

and equation (7):

$$\min_{t \in \{p'_j\}_{j=1}^{n}} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \right\} = 1.$$

First, we handle equation (6). Let $\tau_k \in T$. Since T fully utilizes the processor,

$$\text{there exists a } t \in \{p_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq t$$

Since $p_j = p'_j$ for all $1 \leq j \leq n$,

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e_j \leq t \tag{12}$$

Now we divide our consideration into two subcases based on the value of k in relation to i. Our goal in each case is to show that there exists a $t \in \{p'_j\}_{j=1}^{k}$ such that

$$\sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t,$$

thereby proving T' satisfies equation (6).

Subcase 2.A: $k < n$. By the case 2 assumption, we know that for all τ_j with $1 \leq j \leq k$, $e_j \leq p_{j+1} - p_j$. Additionally, note that the conversions for T' preserve this statement. Namely, for all τ'_j with $1 \leq j \leq k$, $e'_j \leq p'_{j+1} - p'_j$. Let $t = p'_1$. Note that since $t \geq p'_1$ for all τ'_j with $1 \leq j \leq k$, then $\left\lceil \frac{t}{p'_j} \right\rceil = 1$. We then have

$$\begin{aligned}
\sum_{j=1}^{k} \left\lceil \frac{p_k}{p'_j} \right\rceil e'_j &\leq \sum_{j=1}^{k} \left\lceil \frac{p_k}{p'_j} \right\rceil (p_{j+1} - p_j) \\
&= \sum_{j=1}^{k} 1(p_{j+1} - p_j) \\
&= p_{k+1} - p_1
\end{aligned} \tag{13}$$

Now, by the assumptions on period length for this theorem,

$$\begin{aligned}
p_{k+1} &\leq 2p_1 \\
p_{k+1} - p_1 &\leq p_1
\end{aligned} \tag{14}$$

Combining equations (13) and (14), we have

$$\sum_{j=1}^{k} \left\lceil \frac{p_k}{p'_j} \right\rceil e'_j \leq p_1$$

Therefore, for $\tau'_k \in T'$ such that $k < n$,

$$\text{there exists a t } \in \{p'_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

Thus, for subcase 2.A, equation (6) holds.

Subcase 2.B: $k = n$. By assumption, T fully utilizes the processor. Therefore,

$$\text{there exists a } t \in \{p_j\}_{j=1}^{n} \text{ such that } \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j = t$$

We will break our consideration down into two subsubcases based on the value of t.

Subsubcase 2.B.i: $t = p_k \leq p_i$. In this subsubcase, we will show that we cannot satisfy equation (6) for T. The goal is to eliminate this subsubcase from consideration, so that we know equation (6) is satisfied for T from a value of t considered in subsubcase 2.B.ii.

We define p_l such that p_l is the highest indexed period such that $p_l < p_k$. Therefore, we know that for $1 \leq j \leq l$, $\left\lceil \frac{p_k}{p_j} \right\rceil = 2$, and that for $l < j \leq n$, $\left\lceil \frac{p_k}{p_j} \right\rceil = 1$. Knowing these equalities, we have

$$\begin{aligned}
\sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j &= \sum_{j=1}^{l} \left\lceil \frac{p_k}{p_j} \right\rceil e_j + \sum_{j=l+1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j \\
&= \sum_{j=1}^{l} 2e_j + \sum_{j=l+1}^{n} 1e_j \\
&= \sum_{j=1}^{l} 1e_j + \sum_{j=1}^{n} 1e_j \\
&= \sum_{j=1}^{l} e_j + \sum_{j=1}^{n} \left\lceil \frac{p_1}{p_j} \right\rceil e_j \qquad (15)
\end{aligned}$$

Since T satisfies equation (7), then we know that for $t = p_1$,

$$\begin{aligned}
\frac{1}{p_1} \sum_{j=1}^{n} \left\lceil \frac{p_1}{p_j} \right\rceil e_j &\geq 1 \\
\sum_{j=1}^{n} \left\lceil \frac{p_1}{p_j} \right\rceil e_j &\geq p_1 \qquad (16)
\end{aligned}$$

Combining equations (15) and (16), we have

$$\sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j \geq \sum_{j=1}^{l} e_j + p_1$$

And by the assumptions of Case 2, we know that for $1 \leq j \leq l < i$, we have $e_j = p_{j+1} - p_j$. Therefore,

$$\begin{aligned} \sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j &\geq \sum_{j=1}^{l} (p_{j+1} - p_j) + p_1 \\ &= p_{l+1} - p_1 + p_1 \\ &= p_{l+1} \end{aligned}$$

Since p_l is the highest indexed period that is strictly less than p_k, p_{l+1} must be equal to p_k. Therefore, we have

$$\sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j \geq p_k \tag{17}$$

with equality if and only if

$$\begin{aligned} \sum_{j=1}^{n} \left\lceil \frac{p_1}{p_j} \right\rceil e_j &= p_1 \\ \sum_{j=1}^{n} 1 e_j &= p_1 \end{aligned} \tag{18}$$

Recall that we are trying to show that

$$\sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j > p_k$$

and therefore we wish to show that equation (18) is false. Note that by the definitions of Case 2, we have $e_i < p_{i+1} - p_i$. Since $e_j > 0$ for all $1 \leq j \leq n$, then clearly $p_{i+1} > p_i$. Therefore, for all $1 \leq j \leq i$, $\left\lceil \frac{p_{i+1}}{p_j} \right\rceil = 2$, and for all $i < j \leq n$, $\left\lceil \frac{p_{i+1}}{p_j} \right\rceil = 1$. Knowing these equalities, we have

$$\begin{aligned} \sum_{j=1}^{n} \left\lceil \frac{p_{i+1}}{p_j} \right\rceil e_j &= \sum_{j=1}^{i} \left\lceil \frac{p_{i+1}}{p_j} \right\rceil e_j + \sum_{j=i+1}^{n} \left\lceil \frac{p_{i+1}}{p_j} \right\rceil e_j \\ &= \sum_{j=1}^{i} 2e_j + \sum_{j=i+1}^{n} 1e_j \\ &= \sum_{j=1}^{i-1} e_j + e_i + \sum_{j=1}^{n} 1e_j \end{aligned}$$

By the definitions of Case 2, recall that for $1 \leq j < i$, $e_j = p_{j+1} - p_j$ and $e_i = p_{i+1} - p_i - \Delta$. Therefore,

$$\begin{aligned}
\sum_{j=1}^{n} \left\lceil \frac{p_{i+1}}{p_j} \right\rceil e_j &= \sum_{j=1}^{i-1} (p_{j+1} - p_j) + e_i + \sum_{j=1}^{n} e_j \\
&= p_i - p_1 + (p_{i+1} - p_i - \Delta) + \sum_{j=1}^{n} e_j \\
&= p_{i+1} - p_1 - \Delta + \sum_{j=1}^{n} e_j \qquad (19)
\end{aligned}$$

Consider that since T fully utilizes the processor, then by equation (7),

$$\sum_{j=1}^{n} \left\lceil \frac{p_{i+1}}{p_j} \right\rceil e_j \geq p_{i+1}$$

Applying equation (19),

$$\begin{aligned}
p_{i+1} - p_1 - \Delta + \sum_{j=1}^{n} e_j &\geq p_{i+1} \\
\sum_{j=1}^{n} e_j &\geq p_1 + \Delta \qquad (20)
\end{aligned}$$

Which holds for all $p_k \leq p_i$. Equation (17) then becomes

$$\sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j > p_k$$

Therefore we know that

$$\text{for all } t \in \{p_j\}_{j=1}^{n} \text{ with } t \leq p_i, \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j > t \qquad (21)$$

Therefore, for τ_n in T, equation (6) must be satisfied for some $t = p_k > p_i$. That is to say, t must fall into in subsubcase 2.B.ii, since we know equation (6) is true for T.

Subsubcase 2.B.ii: $t = p_k > p_i$. In this subsubcase, we know that $\left\lceil \frac{p_k}{p_i} \right\rceil = 2$. As well, since $p_k \leq p_n$, $\left\lceil \frac{p_k}{p_n} \right\rceil = 1$. With those considerations in mind, we see that

$$\begin{aligned}
\sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j &= \sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e'_j + \left\lceil \frac{p_k}{p_i} \right\rceil e_i + \left\lceil \frac{p_k}{p_n} \right\rceil e_n - \left\lceil \frac{p_k}{p'_i} \right\rceil e'_i - \left\lceil \frac{p_k}{p'_n} \right\rceil e'_n \\
&= \sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e'_j + 2e_i + e_n - 2e'_i - e'_n
\end{aligned}$$

By the definitions of e_i and e'_n, we then have

$$\begin{aligned}
\sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e_j &= \sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e'_j + 2(e'_i - \Delta) + e_n - 2e'_i - (e_n - 2\Delta) \\
&= \sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e'_j + 2e'_i - 2\Delta + e_n - 2e'_i - e_n + 2\Delta \\
&= \sum_{j=1}^{n} \left\lceil \frac{p_k}{p_j} \right\rceil e'_j \\
&= \sum_{j=1}^{n} \left\lceil \frac{p'_k}{p'_j} \right\rceil e'_j \qquad (22)
\end{aligned}$$

Now, since T fully utilizes the processor, by equation (6), we know that

$$\text{there exists a } t \in \{p_j\}_{j=1}^{n} \text{ such that } \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq t$$

However, considering equation (21), the above becomes

$$\text{there exists a } t \in \{p_j\}_{j=1}^{n}, t > p_i \text{ such that } \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq t$$

And since $p_j = p'_j$ for all $1 \leq j \leq n$,

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{n}, t > p'_i \text{ such that } \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e_j \leq t$$

Then by equation (22), we have

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{n}, t > p'_i \text{ such that } \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

Therefore, we know that for $k = n$,

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

Thus, since subsubcase 2.B.i cannot hold, and since equation (6) holds for subsubcase 2.B.ii, then equation (6) holds for subcase 2.B. Additionally, since equation (6) holds for subcase 2.A, then it holds for case 2 as a whole. So, we have shown that for all τ'_k in T',

$$\text{there exists a } t \in \{p'_j\}_{j=1}^{k} \text{ such that } \sum_{j=1}^{k} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

And therefore, in case 2, task set T' satisfies equation (6).

We now prove that, in case 2, T' satisfies equation (7). Since T fully utilizes the processor, then by equation (7)

$$\min_{t \in \{p_j\}_{j=1}^{n}} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1$$

Since we're considering a value where $k = n$, then we may use the results from subsubcase 2.B. Then by equation (21),

$$\min_{t \in \{p_j\}_{j=1}^{n}, t > p_i} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1$$

Since $p_j = p'_j$ for all $1 \leq j \leq n$,

$$\min_{t \in \{p'_j\}_{j=1}^{n}, t > p'_i} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e_j \right\} = 1$$

which, when combined with equation (22), yields

$$\min_{t \in \{p'_j\}_{j=1}^{n}, t > p'_i} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \right\} = 1 \tag{23}$$

Therefore, for equation (7) to hold for T', it remains to prove that

$$\min_{t \in \{p'_j\}_{j=1}^{n}, t \leq p'_i} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \right\} \geq 1$$

First, we will prove a preliminary result...

$$\begin{aligned} \sum_{j=1}^{n} e'_j &= \sum_{j=1}^{n} e_j + e'_i + e'_n - e_i - e_n \\ &= \sum_{j=1}^{n} e_j + (e'_i - e_i) + (e'_n - e_n) \end{aligned}$$

By the case 2 definitions of e_i, e'_i, e_n, and e_n, we then have

$$\begin{aligned} \sum_{j=1}^{n} e'_j &= \sum_{j=1}^{n} e_j + \Delta - 2\Delta \\ &= \sum_{j=1}^{n} e_j - \Delta \end{aligned}$$

By equation (20), we then have

$$\begin{aligned}\sum_{j=1}^{n} e'_j &\geq (p_1 + \Delta) - \Delta \\ &\geq p_1 \end{aligned} \tag{24}$$

Now let us take a look at the sum under consideration for equation (7). Let $t = p'_k \leq p'_i$. Then, by equality of period lengths, we have

$$\begin{aligned}\sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j &= \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e'_j \\ &= \sum_{j=1}^{k-1} \left\lceil \frac{t}{p_j} \right\rceil e'_j + \sum_{j=k}^{n} \left\lceil \frac{t}{p_j} \right\rceil e'_j \end{aligned} \tag{25}$$

We know by the case 2 assumptions that $e_j = p_{j+1} - p_j$ for all $1 \leq j < i$. Since $e_j > 0$ for all such τ_j, then we know that $p_j < p_{j+1}$ for all $1 \leq j < i$. Specifically, we know that since $k \leq i$, for all $j \leq k-1$, $p_j < p_{k-1}$ and for all $j > k-1$, $p_{k-1} < p_j$. Therefore, if $j \leq k-1$, then $\left\lceil \frac{t}{p_j} \right\rceil = 2$ and if $j \geq k$, then $\left\lceil \frac{t}{p_j} \right\rceil = 1$. Thus the equation (25) becomes

$$\begin{aligned}\sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j &= \sum_{j=1}^{k-1} 2e'_j + \sum_{j=k}^{n} 1e'_j \\ &= \sum_{j=1}^{k-1} e'_j + \sum_{j=1}^{n} e'_j \end{aligned}$$

By the case 2 definitions of e_j for $1 \leq j \leq k-1 < i$, $e_j = p_{j+1} - p_j$. Thus we have

$$\begin{aligned}\sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j &= \sum_{j=1}^{k-1} (p_{j+1} - p_j) + \sum_{j=1}^{n} 1e'_j \\ &= p_k - p_1 + \sum_{j=1}^{n} 1e'_j \end{aligned}$$

Combining with equation (24),

$$\begin{aligned}\sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j &\geq p_k - p_1 + p_1 \\ &\geq p_k \end{aligned}$$

Thus, for any $t \in \{p'_j\}_{j=1}^{n}, t \leq p'_i$, we know that

$$\frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \geq 1$$

Combining with equation (23),

$$\min_{t \in \{p'_j\}_{j=1}^{n}} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \right\} = 1$$

which proves that, for case 2, T' satisfies equation (7).

Case 2: Subproof that the utilization of T' is at most that of T: The utilization of the original task set is

$$U = \frac{e_1}{p_1} + \frac{e_2}{p_2} + \cdots + \frac{e_{i-1}}{p_{i-1}} + \frac{p_{i+1} - p_i - \Delta}{p_i} + \frac{e_{i+1}}{p_{i+1}} + \cdots + \frac{e_n}{p_n}$$

The utilization of the modified task set is

$$U' = \frac{e_1}{p_1} + \frac{e_2}{p_2} + \cdots + \frac{e_{i-1}}{p_{i-1}} + \frac{p_{i+1} - p_i}{p_i} + \frac{e_{i+1}}{p_{i+1}} + \cdots + \frac{e_n - 2\Delta}{p_n}$$

Hence, the difference in utilization is

$$\begin{aligned} U - U' &= \frac{-\Delta}{p_i} + \frac{2\Delta}{p_n} \\ &= \frac{2p_i\Delta - p_n\Delta}{p_i p_n} \\ &= \frac{2p_i - p_n}{p_i p_n}\Delta \end{aligned}$$

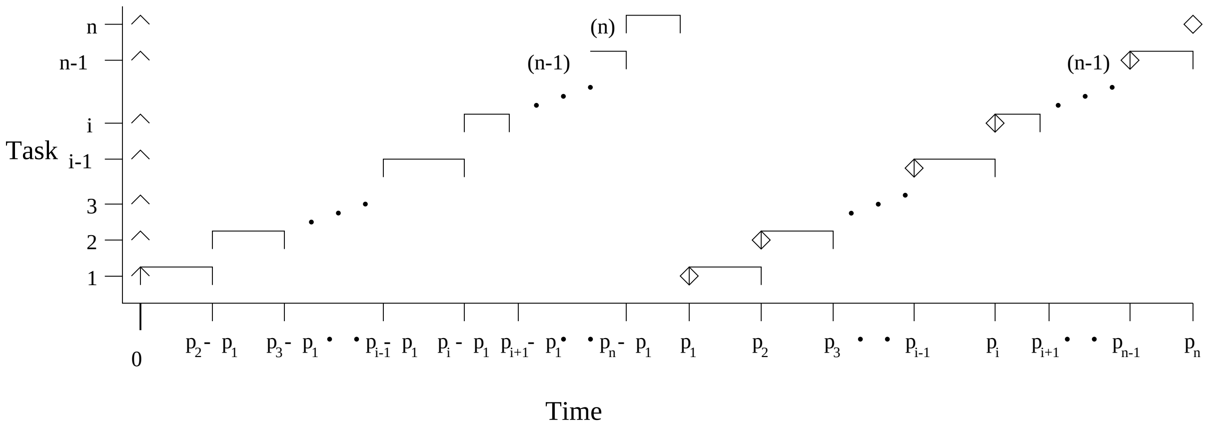
Since Δ, p_i, and p_n are positive, and $2p_i \geq p_n$, we have

$$U - U' = \frac{2p_i - p_n}{p_i p_n}\Delta \geq 0$$

with equality if and only if $2p_i = p_n$. Thus, for case 2, we have provided a task set with a utilization at most that of T that fully utilizes the processor. Additionally, we know that for T', $e'_i \leq p'_{i+1} - p'_i$ for all $i < n$. Thus, T' cannot fall into case 1, and must fall into cases 2 or 3. Also, T' has one less task (than in T) τ'_i such that $e'_i < p'_{i+1} - p'_i$. Since there are a finite number of tasks in the task set, we may apply the case 2 transformation repeatedly, until we know that for all $i < n$, $e_i = p_{i+1} - p_i$. Specifically, repeated transformations will eventually yield a task set whose utilization is at most that of the original task set, that fully utilizes the processor, and which falls into case 3 below.

Case 3: For all $i < n, e_i = p_{i+1} - p_i$.

Here is an example graph of how the schedule would look.



Case 3 sample graph

We must show that this is a nonempty case, and we will also show that the only possible value of e_n is exactly $2p_1 - p_n$. Any other value of e_n will yield a task set that does not fully utilize the processor (a lesser value creates idle time, and a greater value causes an overflow). To do so, we must find the value(s) of e_n such that T satisfies equations (6) and (7). We break our consideration into two cases based on the index of the task in question.

Subcase 3.A: $i < n$. Since we're considering a task other that τ_n, we simply must satisfy equation (6). Namely, we must find a value of $t \in \{p_j\}_{j=1}^{i}$ such that

$$\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq t$$

We let $t = p_1$. Then, since $p_1 \leq p_j$ for all $j = 1, 2, \ldots, i$,

$$\begin{aligned} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j &= \sum_{j=1}^{i} 1 \cdot e_j \\ &= \sum_{j=1}^{i} (p_{j+1} - p_j) \\ &= p_{i+1} - p_1 \end{aligned}$$

Since $p_{i+1} \leq p_n \leq 2p_1$,

$$\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \leq 2p_1 - p_1$$

$$
\begin{aligned}
&= p_1 \\
&\leq p_i
\end{aligned}
$$

and we have the desired result,

$$\sum_{j=1}^{i} \left\lceil \frac{p_1}{p_j} \right\rceil e_j \leq p_1$$

Subcase 3.B: $i = n$. Since we're considering $i = n$, we must find e_n such that equation (7) holds, which then implies that equation (6) holds for $i = n$. We must find e_n such that

$$\min_{t \in \{p_j\}_{j=1}^{n}} \left\{ \frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} = 1.$$

So, let $p_l \in \{p_j\}_{j=1}^{n}$. Let $k \leq n$ such that if $p_1 = p_l$, then $k = 1$ (and we then know that $p_k = p_l$). Otherwise, let k be such that $p_{k-1} < p_l$ and $p_k = p_l$. Then we have $\left\lceil \frac{p_l}{p_j} \right\rceil = 2$ if $j < k$, and $\left\lceil \frac{p_l}{p_j} \right\rceil = 1$ if $j \geq k$. Thus,

$$
\begin{aligned}
\frac{1}{p_l} \sum_{j=1}^{n} \left\lceil \frac{p_l}{p_j} \right\rceil e_j &= \frac{1}{p_l} \left(\sum_{j=1}^{k-1} \left\lceil \frac{p_l}{p_j} \right\rceil e_j + \sum_{j=k}^{n-1} \left\lceil \frac{p_l}{p_j} \right\rceil e_j + e_n \right) \\
&= \frac{1}{p_l} \left(\sum_{j=1}^{k-1} 2e_j + \sum_{j=k}^{n-1} e_j + e_n \right) \\
&= \frac{1}{p_l} \left(\sum_{j=1}^{k-1} 2(p_{j+1} - p_j) + \sum_{j=k}^{n-1} (p_{j+1} - p_j) + e_n \right) \\
&= \frac{1}{p_l} (2(p_k - p_1) + (p_n - p_k) + e_n) \\
&= \frac{1}{p_l} (p_k + p_n - 2p_1 + e_n) \\
&= \frac{1}{p_k} (p_k + p_n - 2p_1 + e_n) \qquad (26)
\end{aligned}
$$

Now we consider possible values of e_n in comparison with $2p_1 - p_n$.

If $e_n < 2p_1 - p_n$, equation (26) becomes

$$\frac{1}{p_l} \sum_{j=1}^{n} \left\lceil \frac{p_l}{p_j} \right\rceil e_j = \frac{1}{p_k} (p_k + p_n - 2p_1 + e_n)$$

$$
\begin{aligned}
&< \frac{1}{p_k}(p_k + p_n - 2p_1 + 2p_1 - p_n) \\
&= \frac{1}{p_k}(p_k) \\
&= 1
\end{aligned}
$$

Since this is true for all $p_l \in \{p_j\}_{j=1}^{n}$, equation (7) fails to hold and T does not fully utilize the processor.

If $e_n > 2p_1 - p_n$, equation (26) becomes

$$
\begin{aligned}
\frac{1}{p_l}\sum_{j=1}^{n}\left\lceil\frac{p_l}{p_j}\right\rceil e_j &= \frac{1}{p_k}(p_k + p_n - 2p_1 + e_n) \\
&> \frac{1}{p_k}(p_k + p_n - 2p_1 + 2p_1 - p_n) \\
&= \frac{1}{p_k}(p_k) \\
&= 1
\end{aligned}
$$

Since this is true for all $p_l \in \{p_j\}_{j=1}^{n}$, equations (6) and (7) fail to hold, and T does not fully utilize the processor.

If $e_n = 2p_1 - p_n$, equation (26) becomes

$$
\begin{aligned}
\frac{1}{p_l}\sum_{j=1}^{n}\left\lceil\frac{p_l}{p_j}\right\rceil e_j &= \frac{1}{p_k}(p_k + p_n - 2p_1 + e_n) \\
&= \frac{1}{p_k}(p_k + p_n - 2p_1 + 2p_1 - p_n) \\
&= \frac{1}{p_k}(p_k) \\
&= 1
\end{aligned}
$$

And therefore equations (6) and (7) hold for τ_n.

Therefore, case 3 is valid only for $e_n = 2p_1 - p_n$, and when that holds, we know that T fully utilizes the processor.

Thus, we know that if our task set falls into case 1 or case 2, then by repeated transformations as defined in those cases, we will arrive at a task set under case 3. The resultant task set has the same period lengths as the original, fully utilizes the processor, and the utilization is at most that of the original task set. We have thus shown that for all task sets with $p_1 \leq p_2 \leq \cdots \leq p_n$ and $\frac{p_n}{p_1} \leq 2$, the execution times $e_i = p_{i+1} - p_i$ for $1 \leq i < n$ and $e_n = 2p_1 - p_n$ minimize utilization. $\square$

It should be noted that if there is some $p_j = p_{j+1}$, then $e'_j = 0$, and the task set $T' = \{(p_{i+1} - p_i, p_i, p_i, 0)\}_{i \in \{1,2,\ldots,j-1,j+1,j+2,\ldots,n-1\}} \cup \{(2p_1 - p_n, p_n, p_n, 0)\}$ has effectively been "pruned" by one task since what would be task τ'_j has an execution time of 0, and is therefore degenerate. In a similar fashion, T' is "pruned" by one task if $2p_1 = p_n$, which would make $e'_n = 0$. Additionally, note that the utilization of T is strictly greater than that of T' unless three conditions hold: 1) $e_i \geq p_{i+1} - p_i$ for all $\tau_i \in T$, 2) for all $\tau_j \in T$ such that $e_j > p_{j+1} - p_j$, $p_j = p_{j+1}$, and 3) $2p_i \neq p_n$. Thus, if those three conditions hold, T has the same utilization of the task set T' (which has less than n tasks) that fully utilizes the processor.

It is in [LL '73]'s case 2 that the proof is faulty. There, case 2 is when there exists some τ_i such that $e_i < p_{i+1} - p_i$, and for all $j < i$, $e_j = p_{j+1} - p_j$. The modified execution times are defined as follows, where $\Delta = (p_{i+1} - p_i) - e_i$.

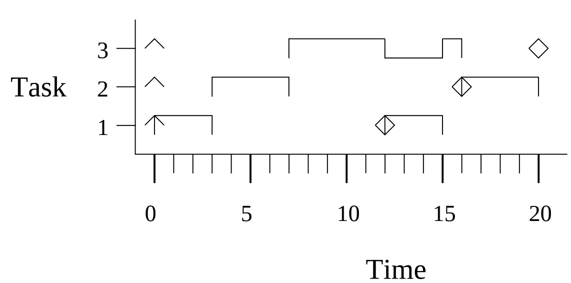
$$\begin{aligned} e'_i &= p_{i+1} - p_i \\ e'_{i+1} &= e_{i+1} - 2\Delta \\ e'_j &= e_j \quad \text{for all} \quad j \neq i, i+1 \end{aligned}$$

The claim is that the modified task set fully utilizes the processor. But consider the following task set T:

$$\begin{aligned} \tau_1 &= (3, 12, 12, 0) \\ \tau_2 &= (4, 16, 16, 0) \\ \tau_3 &= (6, 20, 20, 0) \end{aligned}$$

T fully utilizes the processor since there is no idle time prior to time 20, the latest first deadline. Thus, we have the following valid schedule on $[0, 20)$.

Time	*Task*
$0-3$	τ_1
$3-7$	τ_2
$7-12$	τ_3
$12-15$	τ_1
$15-16$	τ_3
$16-20$	τ_2

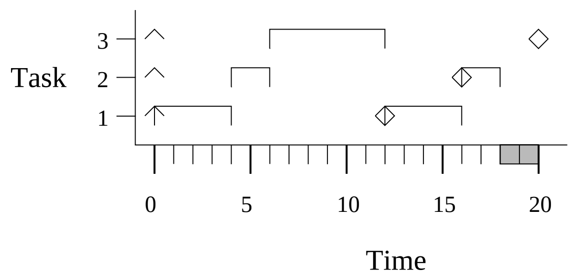


The modified task set then becomes ($\Delta = 1$)

$$\begin{aligned} \tau_1' &= (4, 12, 12, 0) \\ \tau_2' &= (2, 16, 16, 0) \\ \tau_3' &= (6, 20, 20, 0) \end{aligned}$$

and the processor executes as follows

Time	*Task*
$0 - 4$	τ_1'
$4 - 6$	τ_2'
$6 - 12$	τ_3'
$12 - 16$	τ_1'
$16 - 18$	τ_2'
$18 - 20$	idle(!)



Thus, the transformation in case 2 described in [LL '73] does not necessarily produce a task set that fully utilizes the processor. Hence, the induction used in that proof does not hold. Now we expand our consideration to all task sets, and show that the minimum acheived in Theorem 4.2 is a miminum over all task sets.

Lemma 4.9 ([LL '73]) *Let T be a synchronous task set of n tasks that fully utilizes the processor under RM such that there is some task with a period p_i such that $\frac{p_n}{p_i} > 2$. There exists a corresponding synchronous task set T' of n tasks that fully utilizes the processor such that $U(T') \leq U(T)$, and T' has one less task than T where the corresponding ratio of periods is greater than 2.*

We will prove this lemma by generating the task set T' such that T' has one less task (τ_i) such that $\frac{p_n}{p_i} > 2$, and $U(T')$ will be at most $U(T)$.

Proof: Let task τ_i be such that $\frac{p_n}{p_i} > 2$. We will construct T' identical to T except for tasks τ_i and τ_n. Let $q \in \mathbb{Z}_+, r \in \mathbb{R}_+$ be such that $p_n = qp_i + r$, and $0 \leq r < p_i$. Thus, $q \geq 2$. We define τ_i' identically to τ_i, except that $p_i' = qp_i$. We define $p_n' = p_n$, and for the moment we will leave e_n undefined.

We first show that under RM scheduling of T', tasks $\tau'_1, \tau'_2, \ldots, \tau'_{n-1}$ all meet their first deadlines. Consider that for tasks $\tau'_1, \tau'_2, \ldots, \tau'_{i-1}$, no execution times or periods have changed. Additionally, $p_{i-1} \leq p_i < qp_i$, and therefore task τ'_i has a lower priority than any of $\tau'_1, \tau'_2, \ldots, \tau'_{i-1}$. Hence, tasks $\tau'_1, \tau'_2, \ldots, \tau'_{i-1}$ are scheduled in T' exactly as tasks $\tau_1, \tau_2, \ldots, \tau_{i-1}$ are scheduled in T. Since the latter all meet their first deadlines, then we know the former all meet their deadlines.

Let us now consider tasks $\tau'_{i+1}, \tau'_{i+2}, \ldots, \tau'_{n-1}$. Let τ'_j be any of those tasks. Since T fully utilizes the processor, its schedule under RM is valid. Thus, we know that τ_j meets its first deadline in the schedule of T; we denote the time that τ_j completes execution by time t. Then we have

$$\begin{aligned}
\sum_{k=1}^{j} \left\lceil \frac{t}{p_k} \right\rceil e_k &= t \\
\left(\sum_{k=1}^{j} \left\lceil \frac{t}{p'_k} \right\rceil e'_k \right) - \left\lceil \frac{t}{p'_i} \right\rceil e'_i + \left\lceil \frac{t}{p_i} \right\rceil e_i &= t \\
\left(\sum_{k=1}^{j} \left\lceil \frac{t}{p'_k} \right\rceil e'_k \right) - \left\lceil \frac{t}{qp_i} \right\rceil e_i + \left\lceil \frac{t}{p_i} \right\rceil e_i &= t
\end{aligned}$$

Since $qp_i > p_i$, then we know $\frac{1}{qp_i} < \frac{1}{p_i}$ and therefore $\left\lceil \frac{t}{qp_i} \right\rceil \leq \left\lceil \frac{t}{p_i} \right\rceil$. Thus we have

$$\begin{aligned}
\left(\sum_{k=1}^{j} \left\lceil \frac{t}{p'_k} \right\rceil e'_k \right) + \left(\left\lceil \frac{t}{p_i} \right\rceil - \left\lceil \frac{t}{qp_i} \right\rceil \right) e_i &= t \\
\left(\sum_{k=1}^{j} \left\lceil \frac{t}{p'_k} \right\rceil e'_k \right) &\leq t
\end{aligned}$$

Therefore, if τ'_i had the same priority position in T' as does τ_i in T, then τ'_j would meet its first deadline in the schedule of T'. However, τ'_i may have a lower priority than τ_j – but this would mean that τ'_j would satisfy its first release even sooner than time t. Thus, τ'_j will meet its first deadline, regardless of the priority of τ'_i.

Lastly, we must show that τ'_i meets its deadline in the RM schedule of T'. Thus, we must show that Lemma 4.3 is sastisfied. However, it is no longer the case that $P'_1 < P'_2 < \ldots < P'_n$ since the period of task τ'_i has changed. Thus, the consideration is if there exists a $t \leq p'_i$ such that

$$\sum_{P'_j \leq P'_i} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \leq t$$

Since τ_i' is the only task whose period has changed, then we know that $p_1' \leq p_2' \leq \ldots \leq p_{i-1}' \leq p_{i+1}' \leq p_{i+2}' \leq \ldots p_n'$. We let k be such that if there is no $p_j' > p_i'$, $k = n$. Otherwise, we define k such that $p_k' \leq p_i'$ and $p_{k+1}' > p_i'$. Therefore,

$$\{\tau_j | P_j' \leq P_i'\} = \{\tau_j\}_{j=1}^{k}$$

Since τ_k meets its first deadline in the RM schedule of T, there is a $t \leq p_k$ such that

$$\sum_{k=1}^{j} \left\lceil \frac{t}{p_k} \right\rceil e_k = t$$

Now we determine if τ_i' will meet its first deadline:

$$\begin{aligned}
\sum_{P_j' \leq P_i'} \left\lceil \frac{t}{p_j'} \right\rceil e_j' &= \sum_{j=1}^{k} \left\lceil \frac{t}{p_j'} \right\rceil e_j' \\
&= \sum_{j=1}^{i-1} \left\lceil \frac{t}{p_j'} \right\rceil e_j' + \sum_{j=i+1}^{k} \left\lceil \frac{t}{p_j'} \right\rceil e_j' + \left\lceil \frac{t}{p_i'} \right\rceil e_i
\end{aligned}$$

Since $p_i' \geq p_i$, then $\frac{1}{p_i'} \leq \frac{1}{p_i}$, and thus $\left\lceil \frac{t}{p_i'} \right\rceil \leq \left\lceil \frac{t}{p_i} \right\rceil$. Therefore,

$$\sum_{P_j' \leq P_i'} \left\lceil \frac{t}{p_j'} \right\rceil e_j' \leq \sum_{j=1}^{i-1} \left\lceil \frac{t}{p_j'} \right\rceil e_j' + \sum_{j=i+1}^{k} \left\lceil \frac{t}{p_j'} \right\rceil e_j' + \left\lceil \frac{t}{p_i} \right\rceil e_i$$

By definition of each p_j' and e_j', we know that $p_j' = p_j$ and $e_j' = e_j$ for all $j \neq i, n$. That leads to

$$\begin{aligned}
\sum_{P_j' \leq P_i'} \left\lceil \frac{t}{p_j'} \right\rceil e_j' &\leq \sum_{j=1}^{i-1} \left\lceil \frac{t}{p_j} \right\rceil e_j + \sum_{j=i+1}^{k} \left\lceil \frac{t}{p_j} \right\rceil e_j + \left\lceil \frac{t}{p_i} \right\rceil e_i \\
&= \sum_{j=1}^{k} \left\lceil \frac{t}{p_j} \right\rceil e_j \\
&\leq p_k' \\
&= p_k \leq p_i'
\end{aligned}$$

Therefore, task τ_i' meets its deadline in the RM schedule of T'.

Since we know that all tasks $\tau_1', \tau_2', \ldots, \tau_{n-1}'$ meet their deadlines in the RM schedule of T', then there is some value of e_n' (possibly zero) such that T' is schedulable under RM. In fact, there is then some value of e_n' such that T' fully utilizes the processor under RM. That leads us to our definition of e'_n: we define τ'_n identically to τ_n, except that e'_n is set to whatever value fully utilizes the processor under RM for T'.

We begin by showing the following lemma, which will be needed in the remainder of the proof of this theorem.

Lemma 4.10 *Let T and T' be as described above. Then $e_n + (q-1)e_i \geq e'_n$.*

We prove this lemma by contradiction. We will divide our consideration into two cases, based on a time value determined from Lemma 4.7, equation (7).

Proof: Assume otherwise, that $e_n + (q-1)e_i < e'_n$. Since T' fully utilizes the processor, then by equation (7) we know there is some $t \leq p_n$ such that

$$\begin{aligned}
\sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j &= t \\
\left(\sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \right) - \left\lceil \frac{t}{p_i} \right\rceil e_i - \left\lceil \frac{t}{p_n} \right\rceil e_n + \left\lceil \frac{t}{p'_i} \right\rceil e'_i + \left\lceil \frac{t}{p'_n} \right\rceil e'_n &= t \\
\left(\sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \right) - \left\lceil \frac{t}{p_i} \right\rceil e_i - \left\lceil \frac{t}{p_n} \right\rceil e_n + \left\lceil \frac{t}{qp_i} \right\rceil e_i + \left\lceil \frac{t}{p_n} \right\rceil e'_n &= t \qquad (27)
\end{aligned}$$

Since T fully utilizes the processor, then by equation (7),

$$t \leq \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j \qquad (28)$$

Combining equations (27) and (28), we have

$$\begin{aligned}
t - \left\lceil \frac{t}{p_i} \right\rceil e_i - \left\lceil \frac{t}{p_n} \right\rceil e_n + \left\lceil \frac{t}{qp_i} \right\rceil e_i + \left\lceil \frac{t}{p_n} \right\rceil e'_n &\leq t \\
\left\lceil \frac{t}{qp_i} \right\rceil e_i + \left\lceil \frac{t}{p_n} \right\rceil e'_n &\leq \left\lceil \frac{t}{p_i} \right\rceil e_i + \left\lceil \frac{t}{p_n} \right\rceil e_n \\
\left\lceil \frac{t}{p_n} \right\rceil e'_n - \left\lceil \frac{t}{p_n} \right\rceil e_n &\leq \left\lceil \frac{t}{p_i} \right\rceil e_i - \left\lceil \frac{t}{qp_i} \right\rceil e_i \\
\left\lceil \frac{t}{p_n} \right\rceil (e'_n - e_n) &\leq \left(\left\lceil \frac{t}{p_i} \right\rceil - \left\lceil \frac{t}{qp_i} \right\rceil \right) e_i
\end{aligned}$$

By the Lemma's assumption, $(q-1)e_i < e'_n - e_n$, so we have

$$\left\lceil \frac{t}{p_n} \right\rceil (q-1)e_i < \left(\left\lceil \frac{t}{p_i} \right\rceil - \left\lceil \frac{t}{qp_i} \right\rceil \right) e_i$$

$$\left\lceil \frac{t}{p_n} \right\rceil (q-1) \quad < \quad \left\lceil \frac{t}{p_i} \right\rceil - \left\lceil \frac{t}{qp_i} \right\rceil$$

If $t = 0$, then we have

$$0 \cdot (q-1) < 0 - 0$$

which is clearly false. Thus, $0 < t < p_n$, which implies that $\left\lceil \frac{t}{p_n} \right\rceil = 1$, yielding

$$(q-1) < \left\lceil \frac{t}{p_i} \right\rceil - \left\lceil \frac{t}{qp_i} \right\rceil \tag{29}$$

We will now show that equation (29) cannot be satisfied for any $t \in (0, p_n]$ – thereby contradicting our assumption that $e_n + (q-1)e_i < e'_n$.

We divide our consideration into two cases based on the value of t in relation to qp_i.

Case 1: $t \in (0, qp_i]$. Therefore we have

$$\begin{aligned} \left\lceil \frac{t}{qp_i} \right\rceil &= 1 \\ \left\lceil \frac{t}{p_i} \right\rceil &\leq q \end{aligned}$$

which shows

$$\left\lceil \frac{t}{p_i} \right\rceil - \left\lceil \frac{t}{qp_i} \right\rceil \leq q - 1$$

and contradicts Equation (29).

Case 2: $t \in (qp_i, p_n]$. By the definition of q, we know that $qp_i \leq p_n < (q+1)p_i$. Therefore we have

$$\begin{aligned} \left\lceil \frac{t}{qp_i} \right\rceil &= 2 \\ \left\lceil \frac{t}{p_i} \right\rceil &= q + 1 \end{aligned}$$

which shows

$$\left\lceil \frac{t}{p_i} \right\rceil - \left\lceil \frac{t}{qp_i} \right\rceil = (q+1) - 2 = q - 1$$

and contradicts equation (29).

Since we have already eliminated the case where $t = 0$, and cases 1 and 2 cover all possibilities for $t \in (0, p_n]$, then we know that for $t \in [0, p_n]$, equation (29) is false:

$$\sum_{j=1}^{n} \left\lceil \frac{t}{p'_j} \right\rceil e'_j \neq t$$

which contradicts equation (7). However, T' fully utilizes the processor, so equation (7) must hold. So by contradiction of the Lemma assumption,

$$e_n + (q-1)e_i \geq e'_n$$

$\square$

Now back to the main proof of the theorem... The utilization, U', of T' is

$$U' = \sum_{j=1}^{n} \frac{e'_j}{p'_j}$$

Since T and T' only differ in tasks τ_i and τ_n,

$$U' = \left(\sum_{j=1}^{n} \frac{e_j}{p_j} \right) - \frac{e_i}{p_i} - \frac{e_n}{p_n} + \frac{e'_i}{p'_i} + \frac{e'_n}{p'_n}$$

Since $\frac{e'_i}{p'_i} = \frac{e_i}{qp_i}$ and $p'_n = p_n$, we have

$$\begin{aligned} U' &= \left(\sum_{j=1}^{n} \frac{e_j}{p_j} \right) - \frac{e_i}{p_i} + \frac{e_i}{qp_i} - \frac{e_n}{p_n} + \frac{e'_n}{p_n} \\ &= U + \frac{e_i - qe_i}{qp_i} + \frac{e'_n - e_n}{p_n} \end{aligned}$$

Since $qp_i \leq p_n$, then $\frac{1}{qp_i} \geq \frac{1}{p_n}$ and

$$\begin{aligned} U' &\leq U + \frac{e_i - qe_i}{qp_i} + \frac{e'_n - e_n}{qp_i} \\ &= U + \frac{e_i - qe_i + e'_n - e_n}{qp_i} \\ &= U + \frac{e'_n - (e_n + (q-1)e_i)}{qp_i} \end{aligned}$$

By Lemma 4.10, we know that $e_n + (q-1)e_i \geq e'_n$, and therefore $e'_n - (e_n + (q-1)e_i) \leq 0$.

$$U' \leq U$$

Additionally, $\frac{p'_n}{p'_i} < 2$: By definition of p'_i, $p'_i = qp_i + r$ such that $(q+1)p_i > p_n$. Therefore, $2p'_i > 2qp_i \geq (q+1)p_i > p_n = p'_n$. Since $p'_i < p'_n$ and $2p'_i > p'_n$, we have $\frac{p'_n}{p'_i} < 2$.

Thus, we have produced T' as desired: T' fully utilizes the processor and $U(T') \leq U(T)$. $\square$

By repeating this transformation for all tasks $\tau_i \in T$ such that $\frac{p_n}{p_i} > 2$, we produce a sequence of modified task sets, each of which has one less task where $\frac{p_n}{p_i} > 2$, and the utilization of each task set in the sequence is at most the utilization of the previous task sets. We therefore arrive at a task set that has a utilization at most that of the original task set, and the ratios of the periods of the modified tasks are all less than or equal to 2.

4.5 Utilization least upper bound

Having found a minimization based on period length, we now expand our consideration to vary the period lengths. We will derive a minimum utilization over all tasks sets that fully utilize the processor under RM. That utilization value will then determine a break point for considering other task sets: Any task set whose utilization is at most that value must be schedulable.

Theorem 4.3 ([LL '73]) *Over the set of synchronous task sets with n tasks which fully utilize the processor under RM, the minimum utilization is $n(2^{\frac{1}{n}} - 1)$, which is achieved with the values $p_i = 2^{\frac{i-1}{n}} p_1$ for $1 \leq i \leq n$.*

Proof: By Theorem 4.2 and Lemma 4.9, we know for periods $p_1 \leq p_2 \leq \ldots \leq p_n$, the utilization of a task set with those periods is minimized when $\frac{p_n}{p_1} \leq 2$ and the execution times are defined by $e_n = 2p_1 - p_n$, $e_i = p_{i+1} - p_i$ for $1 \leq i < n$. Let us then minimize the processor utilization for such a task set.

$$\begin{aligned} U &= \frac{p_2 - p_1}{p_1} + \cdots + \frac{p_{i+1} - p_i}{p_i} + \cdots + \frac{p_n - p_{n-1}}{p_{n-1}} + \frac{2p_1 - p_n}{p_n} \\ U &= \left(\frac{p_2}{p_1} + \cdots + \frac{p_{i+1}}{p_i} + \cdots + \frac{p_n}{p_{n-1}} + \frac{2p_1}{p_n} \right) - n \qquad (30) \end{aligned}$$

Note that equation (30) may be re-written as

$$U = (x_1 + x_2 + \ldots + x_n) - n \qquad (31)$$

where $x_1 = \frac{p_2}{p_1}, x_2 = \frac{p_3}{p_2}, \ldots, x_{n-1} = \frac{p_n}{p_{n-1}}, x_n = \frac{2p_1}{p_n}$. By the given restrictions on the periods, we then know that for each x_i, $x_i \geq 1$. Additionally, $\prod_{i=1}^n x_i = 2$:

$$\begin{aligned}
\prod_{i=1}^{n} x_i &= \frac{2p_1}{p_n} \prod_{i=1}^{n-1} \frac{p_{i+1}}{p_i} \\
&= \frac{2p_1}{p_n} \cdot \frac{p_2}{p_1} \cdot \frac{p_3}{p_2} \cdot \cdots \cdot \frac{p_{n-1}}{p_{n-1}} \cdot \frac{p_n}{p_{n-1}} \\
&= \frac{2\prod_{i=1}^n p_i}{\prod_{i=1}^n p_i} \\
&= 2
\end{aligned}$$

Since the geometric mean of a set of positive real numbers is less than or equal to the arithmetic mean (see [Ru '66], page 61, for a proof of this claim),

$$\begin{aligned}
\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} &\leq \frac{1}{n}\sum_{i=1}^{n} x_i \\
n(2)^{\frac{1}{n}} &\leq \sum_{i=1}^{n} x_i \\
n(2)^{\frac{1}{n}} - n &\leq \left(\sum_{i=1}^{n} x_i\right) - n
\end{aligned}$$

Then by equation (31)

$$n(2^{\frac{1}{n}} - 1) \leq U$$

Thus, the minimum possible utilization for a task set that fully utilizes the processor under RM is $n(2^{\frac{1}{n}} - 1)$.

To show the origins of the values $p_i = 2^{\frac{i-1}{n}} p_1$, we take the partial of U with respect to p_i:

$$\begin{aligned}
\frac{\partial U}{\partial p_1} &= \frac{2}{p_n} - \frac{p_2}{p_1^2} \\
\frac{\partial U}{\partial p_i} &= \frac{1}{p_{i-1}} - \frac{p_{i+1}}{p_i^2} \qquad \forall \quad 1 < i < n \\
\frac{\partial U}{\partial p_n} &= \frac{1}{p_{n-1}} - \frac{2p_1}{p_n^2}
\end{aligned}$$

Solving each equation for zero, we have

$$2p_1^2 = p_2 p_n$$

$$\begin{aligned} p_i^2 &= p_{i+1}p_{i-1} \qquad \forall \quad 1 < i < n \\ p_n^2 &= 2p_{n-1}p_1 \end{aligned} \tag{32}$$

Note that the second equation above shows that the p_i's form a geometric progression, which can easily be proven by induction: Let a be such that $p_2 = ap_1$. Then for $i = 2$, $p_i = ap_{i-1}$. Now assume that for p_i, $(2 \le i < n)$, $p_i = ap_{i-1}$. Then by equation (32)

$$\begin{aligned} p_i^2 &= p_{i+1}p_{i-1} \\ (ap_{p_{i-1}})^2 &= p_{i+1}p_{i-1} \\ a^2p_{i-1}^2 &= p_{i+1}p_{i-1} \\ a(ap_{i-1}) &= p_{i+1} \\ a(p_i) &= p_{i+1} \end{aligned}$$

Thus, by induction, there is some a such that

$$p_i = p_1 \cdot a^{i-1} \qquad \text{for all } i,\ 1 < i \le n$$

In fact, simple algebra dictates that $p_1 = p_1 \cdot a^{1-1}$, and therefore

$$p_i = p_1 \cdot a^{i-1} \qquad \text{for all } i,\ 1 \le i \le n$$

Thus, we have

$$\begin{aligned} 2p_1^2 &= (p_1a^1)(p_1a^{n-1}) \\ &= p_1^2a^n \\ 2^{\frac{1}{n}} &= a \end{aligned}$$

Therefore,

$$p_i = 2^{\frac{i-1}{n}}p_1 \qquad \forall 1 \le i \le n$$

which also shows $p_1 < p_2 < \ldots < p_n < 2p_1$. By equation (30), the corresponding utilization above becomes

$$\begin{aligned} U &= \left(\frac{p_2}{p_1} + \cdots + \frac{p_{i+1}}{p_i} + \cdots + \frac{p_n}{p_{n-1}} + \frac{2p_1}{p_n}\right) - n \\ &= \left(\frac{2^{\frac{1}{n}}p_1}{p_1} + \cdots + \frac{2^{\frac{i}{n}}p_1}{2^{\frac{i-1}{n}}p_1} + \cdots + \frac{2^{\frac{n-1}{n}}p_1}{2^{\frac{n-2}{n}}p_1} + \frac{2p_1}{2^{\frac{n-1}{n}}p_1}\right) - n \\ &= \left(2^{\frac{1}{n}} + \cdots + 2^{\frac{1}{n}} + \cdots + 2^{\frac{1}{n}} + 2^{\frac{1}{n}}\right) - n \\ &= n2^{\frac{1}{n}} - n \\ &= n(2^{\frac{1}{n}} - 1) \end{aligned}$$

Therefore, the values

$$p_i = 2^{\frac{i-1}{n}} \qquad \text{for all } 1 \le i \le n$$

achieve the minimum utilization for task sets with n tasks that fully utilize the processor under RM. $\square$

See then the corresponding execution times are

$$\begin{aligned}
e_i &= p_{i+1} - p_i \qquad \text{for } 1 \le i < n \\
&= 2^{\frac{i}{n}} p_1 - 2^{\frac{i-1}{n}} p_1 \\
e_n &= 2p_1 - p_n \\
&= 2^{\frac{n}{n}} p_1 - 2^{\frac{n-1}{n}} p_1 \\
\text{Thus,} & \\
e_i &= (2^{\frac{i}{n}} - 2^{\frac{i-1}{n}}) p_1 \qquad \text{for } 1 \le i \le n
\end{aligned}$$

Thus, for any $n \in \mathbb{Z}_+$ and any $p_1 \in \mathbb{R}_+$, there is a task set (which fully utilizes the processor) with a utilization of $n(2^{\frac{1}{n}} - 1)$: the task set $T_{n,p_1} = \left\{ \left((2^{\frac{i}{n}} - 2^{\frac{i-1}{n}}) p_1, 2^{\frac{i-1}{n}} p_1, 2^{\frac{i-1}{n}} p_1, 0 \right) \right\}_{i=1}^{n}$. The significance of T_{n,p_1} is that we have defined the task sets which minimize utilization while fully utilizing the processor. These task sets each have a utilization of $n(2^{\frac{1}{n}} - 1)$. Since they minimize utilization while fully utilizing the processor, then (as we will see below in Theorem 4.4) we know that any task set of n tasks whose utilization is less than $n(2^{\frac{1}{n}} - 1)$ has a valid schedule under RM.

Note that $n(2^{\frac{1}{n}} - 1)$ monotonically decreases in n for $n \ge 1$:

$$\begin{aligned}
2^{\frac{1}{n}} &= e^{\frac{1}{n}\ln 2} \\
&= \sum_{i=0}^{\infty} \frac{\left(\frac{1}{n}\ln 2\right)^i}{i!} \\
&= 1 + \frac{1}{n} \sum_{i=1}^{\infty} \frac{\left(\frac{1}{n}\right)^{i-1} (\ln 2)^i}{i!} \\
2^{\frac{1}{n}} - 1 &= \frac{1}{n} \sum_{i=1}^{\infty} \frac{\left(\frac{1}{n}\right)^{i-1} (\ln 2)^i}{i!} \\
n(2^{\frac{1}{n}} - 1) &= \sum_{i=1}^{\infty} \frac{\left(\frac{1}{n}\right)^{i-1} (\ln 2)^i}{i!} \\
&= \ln 2 + \sum_{i=2}^{\infty} \frac{\left(\frac{1}{n}\right)^{i-1} (\ln 2)^i}{i!}
\end{aligned}$$

Note that every term in the infinite sum montonically decreases in n for $n \geq 1$, thus the sum (and $n(2^{\frac{1}{n}} - 1)$) monotonically decreases in n for $n \geq 1$.

Additionally, the infinite sum summand decreases to 0 as n tends to ∞:

$$\begin{aligned}
\sum_{i=2}^{\infty} \frac{\left(\frac{1}{n}\right)^{i-1} (\ln 2)^i}{i!} &= \frac{1}{n} \sum_{i=2}^{\infty} \frac{\left(\frac{1}{n}\right)^{i-2} (\ln 2)^i}{i!} \\
&< \frac{1}{n} \sum_{i=0}^{\infty} \frac{(\ln 2)^i}{i!} \\
&= \frac{1}{n} e^{\ln 2} \\
&= \frac{2}{n} \\
\lim_{n \to \infty} \sum_{i=2}^{\infty} \frac{\left(\frac{1}{n}\right)^{i-1} (\ln 2)^i}{i!} &= \lim_{n \to \infty} \frac{2}{n} = 0
\end{aligned}$$

Thus, $n(2^{\frac{1}{n}} - 1)$ monotonically decreases to $\ln 2$.

We now summarize the previous results regarding the utilization value $n(2^{\frac{1}{n}} - 1)$.

Theorem 4.4 ([LL '73]) *Under RM,*
1) every synchronous task set of n tasks which satisfies $U \leq n(2^{\frac{1}{n}} - 1)$ is schedulable,
2) there is a schedulable task set of n tasks with $U = n(2^{\frac{1}{n}} - 1)$, and
3) for any value of $U > n(2^{\frac{1}{n}} - 1)$, there exists a task set of n tasks (with such a utilization) that is not schedulable.

Proof of part 1: Let T be a task set of n tasks such that $U(T) \leq n(2^{\frac{1}{n}} - 1)$. We will prove that T has a valid schedule under RM by contradiction. Assume T is not schedulable by RM. Since T is not schedulable, there exists some $\tau_i \in T$ that does not meet its first deadline RM. Let τ_i be the lowest indexed such task. Since lower priority tasks do not affect the scheduling of higher priority tasks under a SPSA, the task set $T' = \{\tau_j\}_{j=1}^{i}$ doesn't fully utilize the processor (by definition of full utilization) because task τ_i misses its first deadline. Additionally,

$$U(T') = \sum_{j=1}^{i} \frac{e_j}{p_j}$$

$$\begin{aligned} &\leq \sum_{j=1}^{n} \frac{e_j}{p_j} \\ &= U(T) \\ &\leq n(2^{\frac{1}{n}} - 1) \end{aligned}$$

Since $i \leq n$ and $n(2^{\frac{1}{n}} - 1)$ decreases monotonically in n, we then have

$$U(T') \leq i(2^{\frac{1}{i}} - 1)$$

Based on this information, we will build a task set T'' that will contradict Theorem 4.3. Thus, T must be schedulable. We define T'' as follows: for all $1 \leq j < i$, let $\tau''_j = \tau'_j$ (which is the same as τ_j). We let $\tau''_i = \tau'_i$, except that we decrease the execution time of e_i such that in RM scheduling of T'', τ''_i meets its first deadline, and there is no idle time prior to p_i. Thus, T'' has either i or $i - 1$ tasks (if e''_i has been set to 0), and all deadlines on $[0, p_i]$ are met. Then, by Lemma 4.5, we know T'' fully utilizes the processor. Now consider the utilization of T'':

$$\begin{aligned} U(T'') &= \sum_{j=1}^{i} \frac{e''_j}{p''_j} \\ &= \sum_{j=1}^{i-1} \frac{e''_j}{p''_j} + \frac{e''_i}{p''_i} \\ &= \sum_{j=1}^{i-1} \frac{e_j}{p_j} + \frac{e''_i}{p_i} \\ &< \sum_{j=1}^{i} \frac{e_j}{p_j} \\ &= U(T') \leq i(2^{\frac{1}{i}} - 1) \end{aligned}$$

Thus, $U(T'') < i(2^{\frac{1}{i}} - 1)$. By the monotonicity of $n(2^{\frac{1}{n}} - 1)$, we also know $U(T'') < (i-1)(2^{\frac{1}{i-1}} - 1)$. Therefore, whether T'' has i or $i-1$ tasks, we have produced a contradiction to Theorem 4.3. Thus, task set T must be schedulable. Therefore, every synchronous task T set of n tasks such that $U(T) \leq n(2^{\frac{1}{n}} - 1)$ is schedulable.

Proof of part 2: As defined immediately after Theorem 4.3, the task sets T_{n,p_1} have utilizations of $n(2^{\frac{1}{n}} - 1)$ and are schedulable under RM. Thus, there is a schedulable task set of n tasks whose utilization is $n(2^{\frac{1}{n}} - 1)$.

Proof of part 3: Let $U > n(2^{\frac{1}{n}} - 1)$. My goal is then to create a task set with a utilization of U such that the task set is not schedulable. We construct T identical to $T_{n,1}$ with one task's execution altered:

$$e_1 = 2^{\frac{1}{n}} - 1 + \Delta$$

where $\Delta = U - n(2^{\frac{1}{n}} - 1)$.

First, we must show that $U(T) = U$:

$$\begin{aligned}
U(T) &= \sum_{i=1}^{n} \frac{e_i}{p_i} \\
&= \frac{e_1}{p_1} + \sum_{i=2}^{n} \frac{2^{\frac{i}{n}} - 2^{\frac{i-1}{n}}}{2^{\frac{i-1}{n}}} \\
&= \frac{2^{\frac{1}{n}} - 1 + \Delta}{2^{\frac{0}{n}}} + \sum_{i=2}^{n} (2^{\frac{1}{n}} - 1) \\
&= \frac{\Delta}{1} + \sum_{i=1}^{n} (2^{\frac{1}{n}} - 1) \\
&= n(2^{\frac{1}{n}} - 1) + \Delta = U
\end{aligned}$$

We will now show that T has no valid schedule under RM by contradicting equation (2) in Lemma 4.4. Since the periods of $T_{n,1}$, and therefore T, follow the descriptions in Lemma 4.8. the set over which we must consider t in equations (6) and (7) is merely $\{p_j\}_{j=1}^{i}$.

Let $t \in \{p_j\}_{j=1}^{n}$. Then

$$\begin{aligned}
\frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j &= \frac{1}{p_l} \left(\sum_{j=1}^{l-1} \left\lceil \frac{p_l}{p_j} \right\rceil e_j + \sum_{j=l}^{n} \left\lceil \frac{p_l}{p_j} \right\rceil e_j \right) \\
&= \frac{1}{p_l} \left(\sum_{j=1}^{l-1} 2e_j + \sum_{j=l}^{n} 1e_j \right) \\
&= \frac{1}{p_l} \left(\sum_{j=1}^{n} e_j + \sum_{j=1}^{l-1} e_j \right)
\end{aligned}$$

If $l = 1$, then we have

$$\begin{aligned}
\frac{1}{t} \sum_{j=1}^{n} \left\lceil \frac{t}{p_j} \right\rceil e_j &= \frac{1}{p_1} (2^{\frac{n}{n}} - 1 + \Delta) \\
&= \frac{1}{1} (2 - 1 + \Delta) \\
&= 1 + \Delta > 1
\end{aligned}$$

If $l > 1$, then we have

$$\begin{aligned}
\frac{1}{t}\sum_{j=1}^{n}\left\lceil\frac{t}{p_j}\right\rceil e_j &= \frac{1}{p_l}\left(\left[\sum_{j=1}^{n}\left(2^{\frac{j}{n}}-2^{\frac{j-1}{n}}\right)+\Delta\right]+\sum_{j=1}^{l-1}\left(2^{\frac{j}{n}}-2^{\frac{j-1}{n}}\right)+\Delta\right)\\
&= \frac{1}{p_l}\left(\left[2^{\frac{n}{n}}-1+\Delta\right]+\left[2^{\frac{l-1}{n}}-1+\Delta\right]\right)\\
&= \frac{1}{2^{\frac{l-1}{n}}}\left(1+\Delta+2^{\frac{l-1}{n}}-1+\Delta\right)\\
&= \frac{1}{2^{\frac{l-1}{n}}}\left(2^{\frac{l-1}{n}}+2\Delta\right)\\
&= \frac{2^{\frac{l-1}{n}}+2\Delta}{2^{\frac{l-1}{n}}} > 1
\end{aligned}$$

Therefore,

$$min_{t\in\{p_j\}_{j=1}^{n}}\left\{\frac{1}{t}\sum_{j=1}^{n}\left\lceil\frac{t}{p_j}\right\rceil e_j\right\} > 1$$

and equation (2) does not hold. Thus, T has no valid schedule under RM. $\square$

4.6 Complexity of feasibility tests

Clearly, there is a polynomial time algorithm that is sufficient to determine if the task set is schedulable – namely, determining if $U \leq n(2^{\frac{1}{n}}-1)$. In practice, one would probably avoid computing $2^{\frac{1}{n}}$ by checking if $\left(\frac{U}{n}+1\right)^{n} \leq 2$. Additionally, we should note that computing utilization of task sets with irrational parameters (and therefore, computing a sum of irrational numbers to compare with a given bound) may not be a polynomial time computation, depending upon the given inputs. However, it is highly probable that all inputs will be rational, making the utilization sum truly a linear time computation. In any event, this test is not necessary for schedulability. We discussed one such necessary and sufficient test above, which is to determine if the first deadline of each task is met. Such an algorithm is pseudo-polynomial, however, since the algorithm must compute the schedule until time $\max\{p_i\}$. The prioritization phase is computable in $O(n\log_2 n)$ time, since one must sort the n priorities. In [LSD '89], another pseudo-polynomial time algorithm is developed that does not require computation of the full schedule from time 0 to time $\max\{p_i\}$, as discussed above. [LSD '89]'s method is based on Lemma 4.4:

$$\max_{\tau_i \in T} \left\{ \min_{t \in \left\{ k \cdot p_j | j \leq i, k \in \{1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\}} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} \right\} \leq 1$$

The given computation indicates that the RM feasibility question is in NP: Given a task set such that RM yields a valid schedule, nondeterministically choose $\{t_i\}_{i=1}^{n}$ such that for all $1 \leq i \leq n$, $0 \leq t_i \leq p_i$ and

$$\frac{1}{t_i} \sum_{j=1}^{i} \left\lceil \frac{t_i}{p_j} \right\rceil e_j \leq 1$$

By the necessary and sufficient nature of Lemma 4.4, we know that such t_i's exist. Each such computation is clearly in $O(n)$, and there are n such computations. Hence, the nondeterministic choices yield a computation time in $O(n^2)$.

To deterministically decide if a task set is schedulable, one would use Lemma 4.4 as well. It is noted in [LSD '89] that the maximum is only necessary for a subset of the given tasks. The idea is that one can use the utilization test defined in Theorem 4.4 to test subsets of the entire task set. Computing the utilization is a linear time computation, and one should make the most of it before switching to that in Lemma 4.4. Namely, one could find the maximal subset of tasks, $\tau_1, \tau_2, \ldots, \tau_m$ (where the tasks are ordered by priority) such that

$$\sum_{j=1}^{m} \frac{e_j}{p_j} < m(2^{\frac{1}{m}} - 1)$$

Since this subset of tasks meets the utilization criterion, then we know this subset will meet all its deadlines – no lower priority task can interfere with the scheduling of these tasks. Hence, the actual computation for equation (2) would be (note the set over which we're now maximizing)

$$\max_{\tau_i \in T, i > m} \left\{ \min_{t \in \left\{ k \cdot p_j | j \leq i, k \in \{1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \right\}} \left\{ \frac{1}{t} \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j \right\} \right\} \leq 1$$

because we already know that tasks $\tau_1, \tau_2, \ldots, \tau_m$ will all meet their first deadline. By Theorem 4.4, we know those tasks already satisfy Lemma 4.3. In practice, one would sort the tasks in increasing order by p_i (as assumed in this section), and step through that sorted task list, summing the $\frac{e_i}{p_i}$'s along the way, and comparing that running sum to $m(2^{\frac{1}{m}} - 1)$. So long as that sum is less than or equal to that bound, then we know that subset of tasks is schedulable (since we know that to be a sufficient, but not necessary, test for schedulability). As soon as the sum became greater than $m(2^{\frac{1}{m}} - 1)$ for m tasks considered, then there would be no guarantee that the m task subset was schedulable. In fact, since $m(2^{\frac{1}{m}}-1)$ decreases as m grows, then every additional task considered will make the utilization (of the subset of tasks) above the $m(2^{\frac{1}{m}}-1)$ bound. Thus, one would have to resort to a test that is sufficient and necessary. Namely, one would use equation (2) for the remaining tasks, as described above. Even with this modification, the algorithm is still pseudo-polynomial.

5 Deadline Monotonic Scheduling and Asynchronous Task Sets

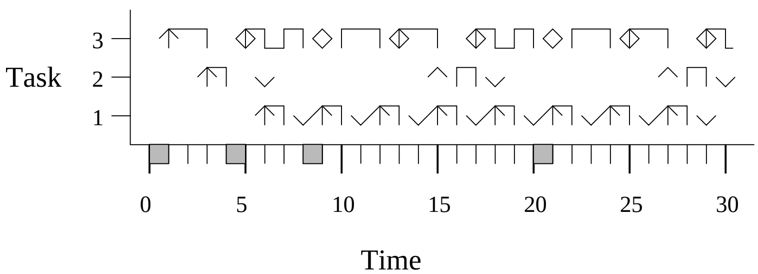
5.1 Definition

Deadline Monotonic scheduling (DM) is a static-priority scheduling algorithm for periodic tasks. DM uses the deadline span of each task for its priority. Thus, tasks with the smallest deadline span will have highest priority, and tasks with the largest deadline span will have the lowest priority. The intuition behind DM is that the task with the smallest deadline span (not necessarily the one with the smallest period) should be the task considered "most urgent," and therefore the task with the highest priority. As in all priority based algorithms, any priority ties may be broken arbitrarily. Formally, DM is a static priority scheduling algorithm with $P_i = d_i$ for all tasks τ_i in the given task set.

5.2 Example

In Section 4, we saw two examples where each task's deadline span was identical to its period. Since RM assigns priorities by period, and DM assigns priorities by deadline span, when the periods are equal to deadline spans, then RM is identical to DM. Thus, the examples in Section 4 are also valid for DM. We present one additional example that will be used later on in this section.

Let T be the task set $\{\tau_i\}_{i=1}^{3}$ such that $\tau_1 = (1, 2, 3, 6)$, $\tau_2 = (1, 3, 12, 3)$, and $\tau_3 = (2, 4, 4, 1)$. By definition of DM, task τ_1 has the highest priority, followed by task τ_2, and then τ_3. It should be noted that under RM, the priorities of τ_2 and τ_3 would be switched. Since no task is released until time 1, the processor is idle at time 0. At time 1, task τ_3 is released, and is the only active task until time 3 – so τ_3 executes to completion. At that time, τ_2 is released, and executes to completion since it is the only active task. No task is active at time 4, so the processor is idle. At time 5, τ_3 is released, and is the only active task – hence it is scheduled until time 6, when task τ_1 is first released. Since τ_1 has a higher priority than τ_3, τ_1 preempts τ_3 and executes at time 6. When τ_1 completes its execution at time 7, τ_3 is no longer preempted (it's now the only active task) and executes. The rest of the graph should be clear.



5.3 DM as an optimal scheduler

As mentioned above, if $p_i = d_i$ for all tasks τ_i in the task set, then DM is identical to RM. Thus, by the work in Section 4.3.3, in those conditions DM is optimal. The benefit derived from DM that is not available in RM is that for synchronous task sets with some $d_j \neq p_j$, DM is optimal, and RM is not. DM is therefore optimal for all synchronous periodic task sets.

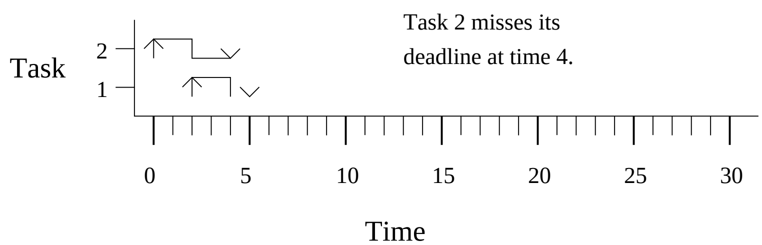
Theorem 5.1 ([LW '82]) *DM is an optimal scheduling algorithm among static priority scheduling algorithms for synchronous task sets.*

Proof: Since task priorities are ordered according to increasing deadline span, this result follows directly from Theorem 4.1. □

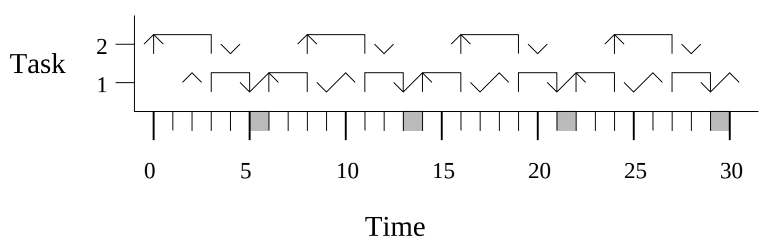
5.4 Asynchronous task sets and a feasibility test

Unfortunately, DM is not optimal for asynchronous task sets. Consider
$T = \{\tau_1 = (2, 3, 4, 2), \tau_2 = (3, 4, 8, 0)\}$. Since τ_1 has the shorter deadline span, DM will assign it a higher priority. Thus, τ_2 will execute on $[0, 2)$, when τ_1 is released. τ_1 will execute on

$[2,4)$ – at which point τ_2 has reached its deadline and has not completed execution. Thus DM does not yield a valid schedule for this task set. However, granting higher priority to task τ_2 will yield a valid schedule: τ_2 will execute on $[0,3)$, τ_1 will execute on $[3,5)$, τ_1 will execute on $[6,8)$, τ_2 will execute on $[8,11)$, τ_1 will execute on $[11,13)$, τ_1 will execute on $[14,16)$, τ_2 will execute on $[16,19)$, and the schedule repeats the pattern defined on $[8,16)$ indefinitely. The graph below shows both the "failed" prioritization and the valid schedule.



Swapping task priorities, we have the following (valid) schedule



According to [LW '82], no static-priority scheduling algorithm has been discovered which is optimal for an arbitrary asynchronous system and produces task prioritizations in polynomial time – we were unable to find either a more recent confirmation of this claim or a development of such an algorithm. Clearly, one means would be to compute all possible prioritizations, and test each one with the feasibility test we will derive in Theorem 5.2. However, simply computing all possible prioritizations requires a factorial (of the number of tasks) amount of time, which doesn't even consider the amount of time it takes to compute feasibility.

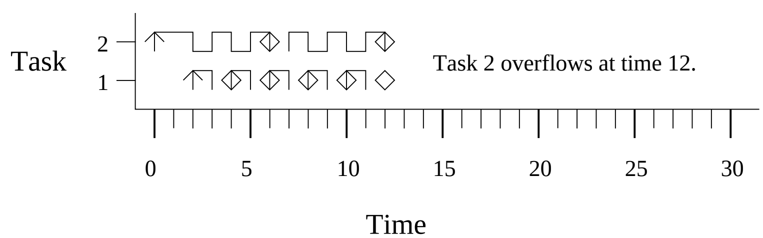
Results from [LW '82] show that DM is also optimal for asynchronous task sets under either of two specific conditions. First, if the task sets under consideration contain only two tasks, and $d_i = p_i$ for $i \in \{1,2\}$. Second, if the task sets are such that $d_i = p_i$ for all $\tau_i \in T$ and for any τ_i, τ_j such that $p_i < p_j$, there exists a $k \in \mathbb{Z}_+$ such that $kp_i = p_j$. We do not duplicate those results here.

[LW '82] then provides a valuable tool, an algorithm to determine schedulability of discrete static priority scheduling algorithms for asynchronous task sets with integer valued parameters. The idea for the algorithm is that under a discrete schedule, the scheduling of the processor will become periodic at time $\max_{\tau_i \in T}\{r_i\} + \text{lcm}\{p_i\}$ (or before). Thus, if all deadlines up to time $\max_{\tau_i \in T}\{r_i\} + 2 \cdot \text{lcm}\{p_i\}$ are met, then all deadlines in the entire schedule are met (since the scheduler will repeat itself after that time). Note that this test does not show DM to be optimal.

We first will make a few definitions and present some necessary preliminary lemmas. We define $r = \max_{\tau_i \in T}\{r_i\}$ and $P = \text{lcm}\{p_i\}$.

To see why we check deadlines up to $r + 2P$ and not just $r + P$, consider the task set $T = \{\tau_1, \tau_2\}$, where $\tau_1 = (1, 2, 2, 2)$, and $\tau_2 = (4, 6, 6, 0)$. Note that $U(T) = \frac{1}{2} + \frac{4}{6} > 1$, and therefore T has no valid schedule. However, using DM, overflow doesn't occur until time 12 – specifically, a timestamp later than $r + P = 2 + 6 = 8$.

$Time$	$Task$	
$0-2$	τ_2	
$2-3$	τ_1	
$3-4$	τ_2	
$4-5$	τ_1	
$5-6$	τ_2	
$6-7$	τ_1	
$7-8$	τ_2	
$8-9$	τ_1	
$9-10$	τ_2	
$10-11$	τ_1	
$11-12$	τ_2	Overflow!



Given a task set with integer valued parameters, the releases and deadlines become periodic after the last release. We will state this claim as a lemma, which will be useful in the ensuing proof.

Lemma 5.1 *Let T be any task set with tasks that have integer valued parameters, τ_i be a task in T, and t_0 be a release of τ_i. As above, we denote* $\max_{\tau_j \in T}\{r_j\} = r$, *and* $P = lcm_{\tau_j \in T}\{p_j\}$. *Then*
1) if $t_0 \in [r, r+P)$, then for all $k \in \mathbb{Z}$ such that $kP + t_0 \geq r_i$, τ_i has a release at time $kP + t_0$,

and
2) if $t_0 \notin [r, r+P)$*, then there exists* $t_1 \in [r, r+P)$ *such that* $(t_1 - t_0) \bmod P \equiv 0$*, and* t_1 *is a release of* τ_i*.*

In essence, this lemma is stating that whatever "happens" on the interval $[r, r+P)$ wholly defines all the releases (and therefore deadlines) for the entire schedule of T.

Proof of part 1: We know that τ_i has releases at all times $r_i + lp_i$ for $l \in \mathbb{Z}_+$. Since t_0 is a release of τ_i, there is some l_0 such that $r_i + l_0 p_i = t_0$. Since $P = \mathrm{lcm}_{\tau_j \in T}\{p_j\}$, then $\frac{P}{p_j}$ is an integer, which we will denote c. Let $k \in \mathbb{Z}$ be such that $kP + t_0 \geq r_i$. Then see that

$$\begin{aligned} r_i + (kc + l_0)p_i &= r_i + kcp_i + l_0 p_i \\ &= r_i + l_0 p_i + kP \\ &= t_0 + kP \end{aligned}$$

Therefore, by definition of k,

$$\begin{aligned} r_i + (kc + l_0)p_i &\geq r_i \\ (kc + l_0)p_i &\geq 0 \\ kc + l_0 &\geq 0 \\ kc + l_0 &\in \mathbb{Z}_+ \end{aligned}$$

Thus, we have found an $l \in \mathbb{Z}_+$ (namely, $kc + l_0$) such that $r_i + lp_i = kP + t_0$. This holds for all $k \in \mathbb{Z}$ such that $kP + t_0 \geq r_i$. For all such k, by definition of release times, τ_i has a release at time $kP + t_0$.

Proof of part 2: Let $t_0 \notin [r, r+P)$ be a release of τ_i. Since t_0 is a release of τ_i, then there exists an $l_0 \in \mathbb{Z}_+$ such that $r_i + l_0 p_i = t_0$. We define $t_1 = r + (t_0 - r) \bmod P$. Clearly, $r \leq t_1 < r + P$. We first must show that $(t_1 - t_0) \bmod P \equiv 0$. By definitions of the variables,

$$\begin{aligned} (t_1 - t_0) \bmod P &\equiv (r + (t_0 - r) \bmod P - t_0) \bmod P \\ &\equiv r \bmod P + t_0 \bmod P - r \bmod P - t_0 \bmod P \\ &\equiv 0 \end{aligned}$$

Since $(t_1 - t_0) \bmod P \equiv 0$, then there exists some $d \in \mathbb{Z}$ such that $t_1 - t_0 = Pd$. Since $P = \mathrm{lcm}_{\tau_j \in T}\{p_j\}$, then $\frac{P}{p_i}$ is an integer, which we denote c. Thus, $t_1 - t_0 = p_i cd$. Then we have

$$\begin{aligned} t_1 &= (t_1 - t_0) + t_0 \\ &= p_i cd + r_i + l_0 p_i \\ &= r_i + (l_0 + cd)p_i \end{aligned}$$

Since $t_1 \geq r$, then $t_1 \geq r_i$. Thus, $l_0 + cd \in \mathbb{Z}_+$, and there exists an l (namely $l_0 + cd$) such that $r_i + lp_i = t_1$. Thus, t_1 is a release of τ_i in $[r, r + P)$ and $(t_1 - t_0) \bmod P \equiv 0$. □

We now move on to make claims about the schedule of a task set, and compare the schedule on the two intervals $[r + P)$ and $[r + P, r + 2P)$. To do so, we will need the following definitions. We define $e_{g,i,t}$ to be the amount of time for which task τ_i executes in schedule g between the release of τ_i immediately prior to (or at) t, and time t. We define $e_{g,i,t} = e_i$ if $t < r_i$ – which indicates that it has no execution pending. Formally, we define $e_{g,i,t}$ as follows: Let $R = \max_{j \in \mathbb{Z}_+}\{r_i + jp_i \leq t\}$. Then

$$e_{g,i,t} = \begin{cases} \int_R^t \chi_{g,\tau_i}(x)dx & : \quad t \geq r_i \\ e_i & : \quad t < r_i \end{cases}$$

Since a task only executes until completion, then for all g, i, t we know that $e_{g,i,t} \leq e_i$. When it is clear, the g subscript will be omitted. Given a schedule g of a task set T, we define $C_g(T, t) = (e_{1,t}, e_{2,t}, \ldots, e_{n,t})$.

Lemma 5.2 ([LW '82]) *Let T be an asynchronous task set with integer parameters. Consider a partial schedule of T for all releases on the interval $[r, r+2P)$ by some static priority scheduling algorithm that meets all deadlines on $(r, r + 2P]$. Then for each task $\tau_i \in T$ and each t such that $r \leq t \leq r + P$, $e_{i,t} \geq e_{i,t+P}$.*

Intuitively, this lemma indicates a relationship between the intervals $[r, r+P)$ and $[r+P, r+2P)$. The relationship is characterized by the fact that a task may not execute any "quicker" on $[r + P, r + 2P)$ than it does on $[r, r + P)$.

Proof: We prove the lemma by contradiction. Assume the lemma is false; that there is some τ_k and some t such that $r \leq t \leq r + P$ and $e_{k,t} < e_{k,t+P}$. Let τ_k be the highest priority task for which there is such a time t. Let R be the release of τ_k immediately prior to (or at) time t. We know such an R exists since $e_{k,t} < e_{k,t+P} \leq e_k$, and $e_{k,t'} = e_k$ for all t' prior to τ_k's first release. Additionally, since we are only considering the partial schedule of T for releases on $[r, r + 2P)$, $R \geq r$. By Lemma 5.1, $R + P$ is the release of τ_k immediately prior to (or at) time $t + P$. Thus, $e_{k,R} = e_{k,R+P} = 0$ since R and $R + P$ are both releases of τ_k. Since $e_{k,t} < e_{k,t+P}$, there is some time in (R, t) where τ_k does not execute, but does execute at the corresponding time on $(R + P, t + P)$. Thus, there exists some time t', $R \leq t' < t$, such that τ_k is not on the processor at time t', but τ_k is on the processor at time $t' + P$. We know that τ_k is active at time t' since $e_{k,t'} < e_{k,t} < e_k$. Therefore, there must be some task τ_l that preempts τ_k at time t' – therefore τ_l has a higher priority than τ_k. Additionally, τ_l must not be active at time $t' + P$ since τ_k, a lower priority task, is on the processor. Thus, τ_l is active at time t', but not at time $t' + P$. Therefore, $e_{l,t'} < e_l = e_{l,t'+P}$. Since $R \geq r$, then we also know that $t' \geq r$. However, this means that we have found a task with higher priority than τ_k such that there exists a time $t' \in [r, r + P]$ such that $e_{l,t'} < e_{l,t'+P}$. This contradicts our assumption that τ_k is the highest priority task with such a time t. Thus, no such task τ_k exists, and the lemma is true. □

We now have a handle on the relation between execution times on the given intervals, and apply that knowledge to derive information about the idle times.

Lemma 5.3 *Let T be an asynchronous task set with integer parameters. Consider any discrete schedule of T. Let $t \geq 0$ be such that $e_{i,t} \geq e_{i,t+P}$ for all $\tau_i \in T$. If the processor is idle at time $t + P$, it must also be idle at time t.*

Proof: Let t be as described above. Then the processor is idle at time $t + P$, and there are no active tasks at time $t + P$. Therefore, for all $\tau_i \in T$, we know that $e_{i,t+P} = e_i$. By the lemma assumption, for all $\tau_i \in T$, $e_{i,t} \geq e_{i,t+P} = e_i$. Thus, $e_{i,t} = e_i$ and τ_i is not active at time t. Since this holds true for all $\tau_i \in T$, the processor is idle at time t. □

We now use Lemma 5.3 to show that the configuration of the schedule at time $r + P$ is identical to its configuration at time $r + 2P$. This result is key in showing that the schedule of a task set is periodic beginning at time $r + P$.

Lemma 5.4 ([LW '82]) *Let T be an asynchronous task set with integer parameters. Consider any discrete (possibly partial) schedule of T that contains all releases on the interval $[r, r + 2P)$ by some scheduling algorithm such that all deadlines on $(r, r + 2P]$ and met, and such that for each task $\tau_i \in T$ and each t such that $r \leq t \leq r + P$, $e_{i,t} \geq e_{i,t+P}$. If the scheduling algorithm is such that offsetting all task releases by P time units yields an identical schedule (offset by P time units), then $C_g(T, r + P) = C_g(T, r + 2P)$.*

To prove the claim, we must show that for each task $\tau_i \in T$, $e_{i,t} = e_{i,t+P}$.

We consider two cases, based on whether there is any idle time on the interval $[r+P, r+2P)$.

Proof:

Case 1: There exists idle time on $[r+P, r+2P)$. Let $t+P \in [r+P, r+2P)$ be such that the processor is idle at time $t+P$. By Lemma 5.1, we know that all releases on $[t+P, r+2P)$ correspond exactly (offset by a value of P) with the releases on $[t, r+P)$. By Lemma 5.3, we know that the processor is idle at time t. Since all tasks are idle at times t and $t+P$, then $C_g(T, t) = C_g(T, t+P)$ (all values are zero). Therefore, the initial values (at t and $t+P$) presented to the scheduler are the same, and all the releases (on $[t, r+P)$ and $[t+P, r+2P)$) are (offset by P) the same. Since the scheduling algorithm is such that offsetting all task releases by P time units yields an identical schedule (offset by P time units), the scheduler then must schedule exactly the same on the intervals $[t, r+P)$ and $[t+P, r+2P)$. Thus, $C_g(T, r+P) = C_g(T, r+2P)$.

Case 2: There is no idle time on $[r+P, r+2P)$. We will use the fact that there is no idle time on this interval to show that for all $\tau_i \in T$, $e_{i,r+P} = e_{i,r+2P}$. Let $\tau_i \in T$, and R_i be the first release of τ_i on the interval $(r+P, r+2P]$. Since τ_i meets all its deadlines on $(r+P, r+2P]$, then it meets its deadline immediately prior to time R_i, which occurs at time $R_i - p_i + d_i$. Therefore, by the definition of $e_{i,t}$, τ_i must execute for $e_i - e_{i,r+P}$ time units on $[r, R_i)$.

On $[R_i, R_i+P-p_i)$, τ_i has $\left\lfloor \frac{P-p_i}{p_i} \right\rfloor = \frac{P}{p_i} - 1$ complete periods. Additionally, $[R_i, R_i+P-p_i) \subset [r+P, r+2P]$: Since R_i is the time of τ_i's first release after time $r+P$, then it cannot be any later than $r+P+p_i$. Thus,

$$\begin{aligned} R_i &\leq r+P+p_i \\ R_i - p_i &\leq r+P \\ R_i + 2P - p_i &\leq r+2P \end{aligned}$$

Therefore, τ_i must meet all its deadlines in $[R_i, R_i+P-p_i]$. There are as many deadlines as there are complete periods, and for each period, there must be e_i time units of execution. Thus, on $[R_i, R_i+P-p_i)$, τ_i executes for $(\frac{P}{p_i} - 1)e_i$ time units.

We now must consider the interval $[R_i + P - p_i, r+2P)$. Since $R_i > r+P$, we know that

$$\begin{aligned} R_i + P - p_i &> r+P+P-p_i \\ (R_i + P - p_i) &> r+2P-p_i \\ (R_i + P - p_i) + p_i &> r+2P \end{aligned}$$

Therefore, the release of τ_i after time $R_i + P - p_i$ falls after time $r+2P$. Thus, the amount of execution that τ_i completes on $[R_i+P-p_i, r+2P)$ is (by the definition of $e_{i,t}$) exactly $e_{i,r+2P}$.

Therefore, τ_i is scheduled for

$$\begin{aligned} e_i - e_{i,r+P} + \left(\frac{P}{p_i} - 1\right) e_i + e_{i,r+2P} &= e_i - e_{i,r+P} + P\frac{e_i}{p_i} - e_i + e_{i,r+2P} \\ &= P\frac{e_i}{p_i} + e_{i,r+2P} - e_{i,r+P} \end{aligned}$$

time units on the entire interval $[r + P, r + 2P)$. So, for all tasks, the amount of total execution on $[r + P, r + 2P)$ is

$$\sum_{i=1}^{n} \left(P\frac{e_i}{p_i} + e_{i,r+2P} - e_{i,r+P}\right)$$

Since there is no idle time on $[r + P, r + 2P)$, we know that the total amount of execution is identical to the interval length. Thus,

$$\begin{aligned} P &= \sum_{i=1}^{n} \left(P\frac{e_i}{p_i} + e_{i,r+2P} - e_{i,r+P}\right) \\ &= P\sum_{i=1}^{n} \frac{e_i}{p_i} + \sum_{i=1}^{n} (e_{i,r+2P} - e_{i,r+P}) \\ &= PU(T) + \sum_{i=1}^{n} (e_{i,r+2P} - e_{i,r+P}) \\ P(1 - U(T)) &= \sum_{i=1}^{n} (e_{i,r+2P} - e_{i,r+P}) \qquad (33) \end{aligned}$$

Note that since P is positive and $U(T) \leq 1$, the left hand side of equation (33) is not negative. Additionally, by the lemma assumption that for each t such that $r \leq t \leq r + P$, $e_{i,t} \geq e_{i,t+P}$, we know the right hand side of equation (33) is not positive, and that the right hand side is zero if and only if $e_{i,r+2P} = e_{i,r+P}$ for all $\tau_i \in T$. Since equation 33 is true, then we must have both sides of the equation equal to zero. Thus, by definition of $C_g(T, t)$, we know that $C_g(T, r + P) = C_g(T, r + 2P)$. □

With the lemmas behind us, we now proceed to prove the main theorem of this section.

Theorem 5.2 ([LW '82]) *Let g be a discrete static priority schedule of T, an asynchronous task set with integer valued parameters. Let g' be the partial schedule of the releases of T on the interval $[r, r + 2P)$ by the same static priority scheduling algorithm used for g. Then g is valid if and only if all deadlines in g' on the interval $(r, r + 2P]$ are met.*

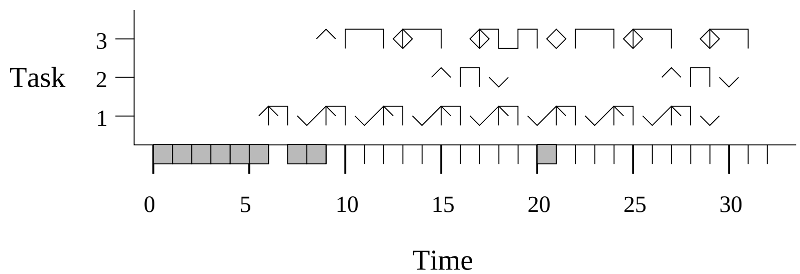
Prior to getting into the proof proper, we note that the theorem assumptions satisfy lemma 5.2, and therefore the theorem assumptions also satisfy lemma 5.3. Additionally, since the schedule is produced by a static priority scheduling algorithm, then offsetting all releases by any time value will not affect prioritization – and therefore not affect how the tasks are scheduled. Thus, the scheduling algorithm is such that offsetting all task releases by P time units yields an identical schedule (offset by P time units), and lemma 5.4 holds.

Proof: We first assume that g is valid, and must show that all deadlines in g' on the interval $(r, r+2P]$ are met. Assume that some deadline in g' is not met. Since the schedule of g contains every release and deadline considered in g', and g' schedules by the same algorithm as g, then the missed deadline in g' must also be missed in g. Informally, g' has "less work" to do on the interval than g has. If g' is unable to meet the deadline, then g certainly won't be able to do so. Therefore, if g is valid, all deadlines in g' on the interval $(r, r+2P]$ are met.

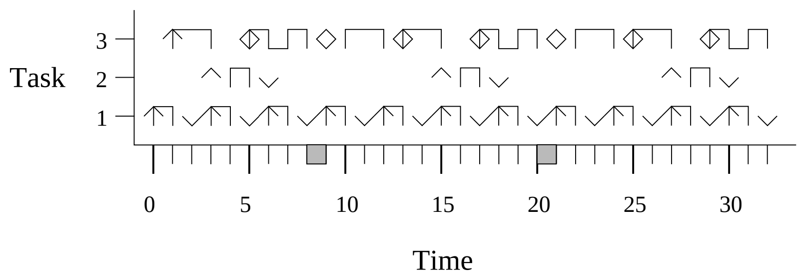
We now assume that in g' all deadlines on $(r, r+2P]$ are met. We will build a schedule, g'', where all deadlines are met, and then compare g'' to g to show that all deadlines in g are met as well. The schedule g'' will be created by "copying" the interval $[r+P, r+2P)$ from g'.

Without loss of generality, assume that $\min_{\tau_i \in T}\{r_i\} = 0$. We build the schedule g'' starting at time $s = r - \left\lfloor \frac{r}{P} \right\rfloor P$ by repeating the schedule interval $[r+P, r+2P)$ in g': That is to say, for any $k \in \mathbb{Z}_+$, g'' on $[s+kP, s+(k+1)P]$ is identical to g' on $[r+P, r+2P]$. We complete the schedule g'' on $[0, s)$ by repeating the schedule interval $[r+2P-s, r+2P)$ from g'. However, there may be execution in g'' on $[0, r+P)$ corresponding to releases that do not occur in T. Thus, we finalize g'' by replacing all such execution time with idle time.

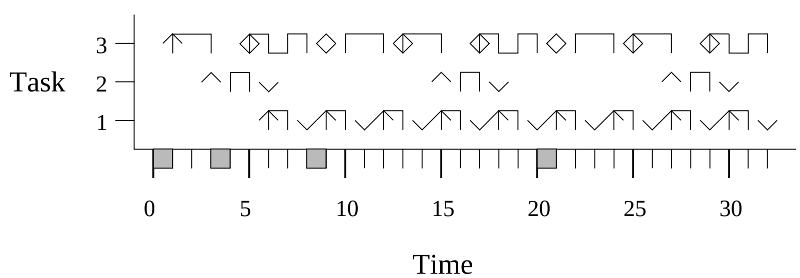
To help visualize how these schedules are related, here are the DM schedules of g, g', and g'' for the sample task set $T = \{\tau_i\}_{i=1}^{3}$ such that $\tau_1 = (1, 2, 3, 6)$, $\tau_2 = (1, 3, 12, 3)$, and $\tau_3 = (2, 4, 4, 1)$ on the interval $[0, 32]$. Note that in this example, $r = 6$, and $P = 12$. Therefore, $r + P = 18$, and $r + 2P = 30$.



The partial schedule g’ of task set T on [0,32]

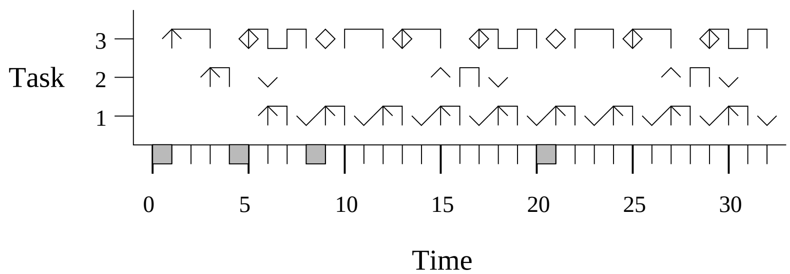


The schedule g’’ of task set T produced by copying
the interval [18,30) from g’ before removal of
extraneous execution.



The schedule g’’ of task set T
(after removal of extraneous execution)

And here is the schedule g of T. Note the minor difference between g and g'' – namely, at times 3 and 4.



The schedule g of task set T on [0,32]

Now back to the proof...

By Lemma 5.4, $C_{g'}(T, r+P) = C_{g'}(T, r+2P)$, and we know by the theorem assumption that all deadlines are met in g' on $(r+P, r+2P]$. Since for times at or after $r+P$, g'' is created by repeating the interval $[r+P, r+2P]$ from g', there can be no missed deadlines in g'' at or after time $r+P$. Additionally, for times before $r+P$, g'' is created by repeating the same interval from g', but with some execution (possibly) removed. The removal of execution time will not delay any other execution, and therefore there can be no missed deadlines in g'' on $[0, r+P)$. Thus, g'' is a valid schedule of T. Additionally, all deadlines and releases are identical to those in g by lemma 5.1.

Now we compare the schedule g'' to the schedule g. We know that g'' has the same releases and deadlines as those in g, and that all deadlines are met in g''. Therefore, if we can show that $e_{g,i,t} \geq e_{g'',i,t}$ for all $\tau_i \in T$ and all $t \geq 0$, then all deadlines in g are met (by applying the result to the time and task of each deadline). We do this by contradiction.

Assume there is some time $t \geq 0$ and some τ_i such that $e_{g,i,t} < e_{g'',i,t}$. Since neither schedule has scheduled any task by time zero, we know $t > 0$. Without loss of generality, let t be the minimal time such that there is a task τ_k with $e_{g,k,t} < e_{g'',k,t}$. Since we are dealing with discrete units of time, we know that this minimum is attained. Additionally, we know that the priorities used in g and g'' are identical, and therefore discussing task priorities is not dependent on a particular schedule. Without loss of generality, we then assume that τ_k is the highest priority task such that $e_{g,k,t} < e_{g'',k,t}$. Therefore, there is either 1) some task τ_l with a higher priority than τ_k that preempts τ_k at time t in g, but does not preempt τ_k at time t in g'', or 2) g schedules no task at time t. Since g schedules by (unaltered) DM, we know that g cannot be idle at t since task τ_k is active. Therefore, there must be a τ_l as described. Since priorities in g and g'' are identical, then τ_l must be active in g at time t and not active in g'' at time t. Therefore, $e_{g,l,t} < e_l$ while $e_{g'',l,t} = e_l$. Thus, τ_l is such that $e_{g,l,t} < e_{g'',l,t}$, which contradicts our assumption about τ_k. Therefore, there can be no such time t, and $e_{g,i,t} \geq e_{g'',i,t}$ for all $\tau_i \in T$ and all $t \geq 0$. $\square$

5.5 Complexity of feasibility tests

Clearly, since DM is a more general case of RM, then feasibilty tests for DM will be at least as complex as those of RM. In the synchronous case, by lemma 4.2, we have a pseudo-polynomial time algorithm to determine if the schedule produced by DM is valid: Produce the schedule until time $\max_{\tau_i \in T}\{d_i\}$. If all first deadlines are met, then the schedule is valid.

Additionally, we have seen that for a given static priority scheduling algorithm, we have a feasibility test, as shown in Theorem 5.2. Note, however, that the feasibility test there is polynomial in n and in the least common multiple of $\{p_i\}_{i=1}^n$. In particular, it is *not* polynomial in n and $\max\{p_i\}_{i=1}^n$ since the least common multiple may very well be on the order of $\prod_{i=1}^n p_i$, which can approach $(p_1)^n$. Thus, we do not know of a pseudo-polynomial time algorithm for feasibility in the asynchronous case.

In fact, given some task set T, determining if there is a valid static-priority schedule of T is co-*NP*-complete in the strong sense. This is shown in [LW '82] by a reduction of the Simultaneous Congruences Problem (SCP) to the feasibility problem above. SCP has been shown to be *NP*-complete in the strong sense in [BHR '93].

First, we define SCP: Given n ordered pairs of positive integers $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ and a positive integer $K (2 \leq K \leq n)$, is there a subset of $l \geq K$ ordered pairs $(a_{i_1}, b_{i_1}), (a_{i_2}, b_{i_2}), \ldots, (a_{i_l}, b_{i_l})$ such that there is a positive integer x such that $x \equiv a_{i_j} \mod b_{i_j}$ for each $1 \leq j \leq l$?

Now, the reduction. Given an instance of SCP, $(a_1, b_1), \ldots, (a_n, b_n)$ and K, we construct the following task system, T, of n tasks: for all $i, 1 \leq i \leq n, \tau_i = (1, K-1, (K-1)b_i, (K-1)a_i)$. Since each task has a computation time of 1, a deadline span of $K-1$, and release times and periods that are multiples of $K-1$, then an overflow will occur if and only if K (or more) tasks are released at a given multiple of $K-1$. By simple algebra, task τ_i is released at time x if and only if $x \equiv (K-1)a_i \mod (K-1)b_i$. Hence, there is overflow if and only if there is some positive integer x and there are $l \geq K$ tasks $\{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_l}\} \subseteq \{\tau_j\}_{j=1}^n$ such that $x \equiv (K-1)a_{i_k} \mod (K-1)b_{i_k}$ for all $1 \leq k \leq l$. Therefore x is a multiple of $K-1$, and for $y = \frac{x}{K-1}$, $y \equiv a_{i_k} \mod b_{i_k}$. Note that this condition on y is exactly the condition for a solution to SCP. Clearly this reduction is polynomial in time, so if there exists a polynomial time algorithm to determine if a task set is not schedulable on a uniprocessor system, then there exists a polynomial time algorithm to solve SCP. Since SCP is *NP*-complete, determining if a task set is **not** schedulable on a uniprocessor is *NP*-hard. Thus, the feasibility problem (determining if a task set is schedulable on a uniprocessor) is co-*NP*-hard.

Now we must show that the feasibility problem is in co-*NP*. Consider any task set that does not have a valid schedule under the given static priority scheduling algorithm. By Theorem 5.2, we know that the partial schedule of all task releases on $[r, r+2P)$ will then have a missed deadline on $(r, r+2P]$. If a deadline is missed, then the amount of execution requested over a given amount of time is greater than the amount of time available. Let us assume that, in the partial schedule, overflow occurs at time t_2. Then there must be some time $t_1 \geq R$ such that there is no idle time on $[t_1, t_2)$. We define t_1 as the minimum over all such times. Therefore, the processor is idle prior to t_1. Consider any task τ_i in the task set. We know that the index of τ_j's release immediately prior to t_2 is $\left\lceil \frac{t_2 - r_j}{p_j} \right\rceil$, and that the index of τ_j's release immediately prior to t_1 is $\left\lceil \frac{t_1 - r_j}{p_j} \right\rceil$. Therefore, τ_j has exactly $\left\lceil \frac{t_2 - r_j}{p_j} \right\rceil - \left\lceil \frac{t_1 - r_j}{p_j} \right\rceil$ releases on $[t_1, t_2)$. Thus, if there is an overflow at time t_2 by some task τ_i, then we know that the amount of work requested by τ_i and all higher priority tasks on the interval $[t_1, t_2)$ is greater than the amount of time available. Namely,

$$\sum_{j=1}^{i} \left(\left\lceil \frac{t_2 - r_j}{p_j} \right\rceil - \left\lceil \frac{t_1 - r_j}{p_j} \right\rceil \right) e_j > t_2 - t_1$$

Thus, given a task set without a valid schedule, there must exist such times t_1 and t_2. To see that the feasibility problem is in co-*NP*, we simply choose (non-deterministically) the appropriate t_1 and t_2. The computation above is polynomial in time, and confirms that the task set has no valid schedule. Thus, the feasibility question is in co-*NP*.

Since the general feasibility problem is co-*NP*-hard and in co-*NP*, it is co-*NP*-complete.

6 Earliest Deadline First

Earliest-deadline-first scheduling (EDF) is one of the most significant scheduling algorithms in the field. The main reason is that EDF is optimal for scheduling any task set. In fact, for task sets where each task's period is identical to its deadline span, EDF will produce a valid schedule if and only if the utilization of the task set is one or less. We've already seen that if a task set has a utilization over one that the task set has no valid schedule, so the power of EDF is the wide range of task sets over which EDF is optimal.

6.1 Definition

EDF is a dynamic-priority scheduling algorithm that assigns highest priority to whatever task has the "nearest deadline". Formally, a task τ_i's priority at time t is given by

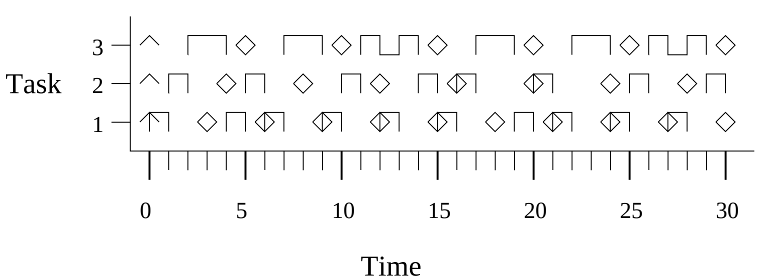
$$P_i = d_i(t) - t$$

where $d_i(t)$ is the next deadline of τ_i (at or after t).

6.2 Example

As we will see below, EDF is able to schedule task sets that RM is unable to schedule. For example, in Section 4 we stated that for the task set $\{(1,3,3,0),(1,4,4,0),(1,5,5,0)\}$, the third task's execution could not be increased if RM were to produce a valid schedule for the task set. However, this is not the case for EDF. Consider the task set $T = \{\tau_i\}_{i=1}^{3}$, where $\tau_1 = (1,3,3,0)$, $\tau_2 = (1,4,4,0)$, and $\tau_3 = (2,5,5,0)$. Since ties in priority may be broken arbitrarily, we will schedule this task set with EDF where, in the case of a priority tie, the task set with the lowest index is scheduled.

At time 0, all three tasks are active, but τ_1 has the nearest deadline (at time 3). Thus, τ_1 is scheduled at 0. At time 1, τ_2 has a nearer deadline than τ_3, so τ_2 is scheduled at time 1. At time 2, τ_3 is the only active task and therefore is scheduled. At time 3, τ_1 is also active, but τ_3's deadline is at 5, whereas τ_1's deadline is at 6 – so τ_3 is scheduled at time 3 and completes, satisfying its deadline at 5 (in RM, τ_1 would have preempted τ_3 and there would have been a missed deadline). At time 4, τ_2 is active, but τ_1 has a nearer deadline and is therefore scheduled. A continuation of this process completes the schedule. A couple of times of interest are at time 12, where τ_1 preempts τ_3 since their corresponding deadlines are identical – so a different choice of how to break ties might yield a different scheduled task at time 12; and at time 18, where task τ_3 preempts task τ_1 since τ_3's deadline is nearer.



6.3 EDF as an optimal scheduler

As mentioned above, EDF is an optimal scheduler for task sets where each task's period is identical to its deadline span. In fact, we will show that EDF is optimal for all task sets. The distinction of task sets where periods equal deadline spans is worth invesitigating, however, because for those task sets, there is a very simple (necessary and sufficient) means of determining schedulability – is the utilization of the task set one or less? If so, the task set is schedulable by EDF, as we will now see.

Theorem 6.1 ([LL '73]) *For task sets T where $p_i = d_i$ for all $\tau_i \in T$, EDF produces a valid schedule if and only if the given task set has a utilization less than or equal to 1.*

Note that we are not assuming that $r_i = 0$ for all i, as is assumed in [LL '73].

Proof: Assume that for some task set T, the schedule produced by EDF is not feasible. Then there must exist some time t_2 when overflow occurs. Let us assume that task τ_j overflows at t_2. Thus, τ_j has a deadline at t_2, and we assume that $t_2 > 0$ (if $t_2 = 0$, then $d_j = 0$, and task τ_j is degenerate). Since τ_j overflows at t_2, then τ_j is active on the entire interval $[t_2 - d_j, t_2)$. Thus, there can be no idle time on that interval (otherwise τ_j would be executing in it!). Therefore, some task(s) is(are) executing on the processor on $[t_2 - d_j, t_2)$. Let t_1 be the time such that $t_1 = 0$ or there is some $\epsilon > 0$ such that the processor is idle on $[t_1 - \epsilon, t_1)$, and there is no idle time on $[t_1, t_2)$. Note that $t_2 - t_1 > 0$ because the processor is not idle on $[t_2 - d_j, t_2)$. Additionally, for some portion of the interval $[t_2 - d_j, t_2)$, τ_j must be preempted by another task with the same or higher priority, and may only be preempted such a task – hence, the preempting task(s) must have a deadline at or before t_2. In other words, no task invocation whose deadline is after t_2 will preempt τ_j for the release at time $t_2 - d_j$. Note that this situation is very different from RM or DM, since proximity to deadlines has no effect on prioritization in those schemes. It is for that reason that this proof does not hold for those scheduling policies.

Let $t_2 - t_1 = t$. By the explanation above, we know $t \geq t_2 - d_j > 0$. Since there is overflow at time t_2 and from t_1 to t_2, the processor is not idle, then the time required for the amount of work requested from t_1 to t_2 is greater than the time available. Knowing that the number of releases (that must be satisfied) of any task τ_i in $[t_1, t_2)$ is at most $\left\lfloor \frac{t}{d_i} \right\rfloor$, that there is no idle time from t_1 to t_2, and that there is overflow at time t_2, we know that the amount of execution requested on $[t_1, t_2)$ is more than the amount of time available, namely:

$$\sum_{i=1}^{n} \left\lfloor \frac{t}{d_i} \right\rfloor e_i \quad > \quad t$$

$$\sum_{i=1}^{n} \left(\frac{t}{d_i}\right) e_i > t$$
$$t \sum_{i=1}^{n} \frac{e_i}{d_i} > t$$
$$\sum_{i=1}^{n} \frac{e_i}{d_i} > 1$$

Thus, if EDF produces an invalid schedule for some task set such that for all $i, d_i = p_i$, that task set has a utilization greater than 1. Hence, if the task set has a utilization of 1 or less, then EDF will produce a valid schedule.

Combining the result above with Theorem 3.1, we have the desired result: EDF produces a valid schedule if and only if the utilization of the given task set is less than or equal to 1. □

Since any task set with a utilization greater than 1 is not schedulable (as shown by Theorem 3.1), then EDF is optimal in the case where $d_i = p_i$ for all i. Note that we have a linear time feasibilty test – merely discern the utilization of the task set (as noted in Section 4.6, the computation of the utilization may not be computable in polynomial time if the task set has irrational parameters).

If there exists some τ_i such that $d_i \neq p_i$, then the above proof still holds, but the feasibility test no longer considers the utilization of the task set: the denominators of the summands are not the periods of the tasks. For example, consider the task set {(1,1,4,0), (1,1,4,0)}. The utilization is $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$, and yet

$$\sum_{i=1}^{n} \frac{e_i}{d_i} = \frac{1}{1} + \frac{1}{1} = 2 > 1$$

And this task set is not schedulable via EDF because both tasks require one unit of execution by time one. However, the task set {(1,1,4,0), (1,1,4,2)} yields the same computations as above, and yet this task set is schedulable (the first task executes at times $0, 4, 8, 12, \ldots$, while the second executes at $2, 6, 10, 14, \ldots$). This example shows that the test of $\sum_{i=1}^{n} \frac{e_i}{d_i}$ is sufficient for schedulabilty, but not necessary. It is solely when this sum is identical to computing utilization that we have a necessary and sufficient test.

So, the question remains: Is EDF optimal for task sets whose task periods are not identical to their deadline spans?

Theorem 6.2 ([La '74]) *EDF is an optimal scheduling algorithm for all task sets.*

In this case, we must show that if there exists a valid schedule for a given task set, then EDF produces a valid schedule.

Proof: As in the proof of Theorem 6.1, let us assume that there is some task set T that is not schedulable via EDF. We define t_2 and t_1 in the same manner – t_2 is the time of the first missed deadline, and t_1 is either 0, or the last time prior to t_2 such that the processor is idle immediately prior to t_1. As above, $t_1 < t_2$. By the definition of t_1 and t_2, any task scheduled in $[t_1, t_2)$ must correspond to a release of that task in $[t_1, t_2)$ (since the processor is idle prior to t_1, then there can be no active task immediately prior to t_1). As well, any task deadline prior to t_2 is met. Lastly, there is a task release in $[t_1, t_2)$ whose deadline is not met (namely, at t_2). Let δ be the amount of time scheduled in $[t_1, t_2)$ for task invocations with deadlines at t_2. Since the processor is never idle in $[t_1, t_2)$, then the amount of time scheduled in $[t_1, t_2)$ for task invocations with deadlines prior to t_2 is exactly $t_2 - t_1 - \delta$. Suppose there is an algorithm, A, that produces a valid schedule for T. Then, in $[t_1, t_2)$, A must devote at least $t_2 - t_1 - \delta$ time units to task invocations whose deadlines are prior to t_2. Additionally, in $[t_1, t_2)$, A must devote more than δ time units to tasks invocations with deadlines at t_2 – otherwise, an overflow will occur at t_2. Since $t_2 - t_1 - \delta + \delta = t_2 - t_1$, it is impossible for A to schedule more than δ time units to those task invocations. Hence, A will overflow at t_2. Thus, if EDF cannot schedule the task set, neither can any other scheduling algorithm. □

6.4 A feasibility test

Note that much of this work parallels work in Section 5.4, and we are able to use the lemmas there to greatly simplify our efforts here.

[LM '80] derives an algorithm to determine the feasibility of producing a valid schedule under EDF for asynchronous task sets with integer valued parameters. The idea for the algorithm is that under a discrete schedule, the scheduling of the processor will become periodic at time $\max_{\tau_i \in T}\{r_i\} + \text{lcm}\{p_i\}$ (or before). Thus, if all deadlines up to time $\max_{\tau_i \in T}\{r_i\} + 2 \cdot \text{lcm}\{p_i\}$ are met, then all deadlines in the entire schedule are met (since the scheduler will repeat itself after that time).

Prior to the proof of this claim, we first will recall a few definitions and present some preliminary lemmas. $e_{g,i,t}$ is defined as the amount of time for which task τ_i has executed in schedule g since its last request up until time t. $e_{g,i,t} = e_i$ if $t < r_i$. When it is clear, the g subscript will be omitted. Given a schedule g, $C_g(T, t) = (e_{1,t}, e_{2,t}, \ldots, e_{n,t})$. As in Section 5.4, we define $r = \max_{\tau_i \in T}\{r_i\}$ and $P = \text{lcm}\{p_i\}$.

Lemma 6.1 ([LM '80]) *Let T be an asynchronous task set with integer parameters. Let g be the discrete schedule of T produced by EDF. For each task $\tau_i \in T$ and each $t \geq r_i$, $e_{i,t} \geq e_{i,t+P}$*

Proof: Assume otherwise, that there is some τ_{k_1} and some $t \geq r_{k_1}$ such that $e_{k_1,t} < e_{k_1,t+P}$. Then there must be some time t' such that $r_{k_1} \leq t' < t$, τ_{k_1} is active at both t' and $t' + P$, and τ_{k_1} is scheduled at time $t' + P$ but not at t'. Thus, there is some task τ_{k_2} with a nearer deadline than τ_{k_1} that is active at time t' and not active at time $t' + P$. Thus, $e_{k_2,t'} < e_{k_2,t'+P}$. By repeating the above argument, then we see that there must be another such task τ_{k_3} which has a nearer deadline than that of τ_{k_2}, then another such task τ_{k_4} with a nearer deadline than that of τ_{k_3}, and so on. Since there are only a finite number of tasks in the task set, then such an infinite sequence cannot exist. Hence, no such τ_{k_1} exists. $\square$

Lemma 6.2 ([LM '80]) *Let T be an asynchronous task set with integer parameters. Let g be the discrete schedule of T produced by EDF, and assume that g meets all deadlines on the interval $(r + P, r + 2P]$. Then $C_g(T, r + P) = C_g(T, r + 2P)$.*

Proof: By lemma 6.1, we know that for each task $\tau_i \in T$ and each t such that $r \leq t \leq r+P$, $e_{i,t} \geq e_{i,t+P}$. By the lemma assumption, g is a discrete schedule that contains all releases on the interval $[r, r + P)$ and meets all deadlines on the interval $(r + P, r + 2P]$. Additionally, the scheduling of EDF is not affected by offsetting all release times by the same amount – since when releases are offset, so are the corresponding deadlines. EDF prioritizes by the difference between the given time and respective deadline, so an offset will not change the prioritization produced by EDF. Thus, lemma 5.4 holds, and $C_g(T, r + P) = C_g(T, r + 2P)$.

$\square$

Theorem 6.3 ([LM '80]) *Let g be the schedule of T, an asynchronous task set with integer valued parameters, produced by EDF. g is a valid schedule if and only if all deadlines in the interval $[0, r + 2P]$ are met.*

Proof: Assume g is valid. Then all deadlines in g are met, including those on $[0, r + 2P]$.

Assume all deadlines in $[0, r+2P]$ are met. Then by lemma 6.2, we know that $C_g(T, r+P) = C_g(T, r + 2P)$. By the same explanation in lemma 6.2, we know that offsetting task releases by a value of P will yield the same schedule (offset by P) under EDF. By lemma 5.1, we know that all releases and deadlines correspond exactly to those on $[r, r+P)$. Thus, we know that for any $k \in \mathbb{Z}_+$, the schedule on $[r+kP, r+(k+1)P)$ is identical to that on $[r+P, r+2P)$ – where all deadlines are met. Therefore, all deadlines at or after time $r+2P$ are met. By the lemma assumption, all deadlines in $[0, r+2P]$ are met. Therefore, all deadlines of T in g are met, and g is valid. $\square$

[BRH '90] followed the work of [LM '80], and produced another feasibility test for EDF which does not require one to compute the entire schedule on the interval $[0, r+2P]$. Prior to stating their claim, we first define $\eta_i(t_1, t_2)$ to be the total number of natural numbers k such that

$$\begin{aligned} t_1 &\leq r_i + kp_i && \text{(a release occurs at or after time } t_1\text{) and} \\ r_i + kp_i + d_i &\leq t_2 && \text{(its corresponding deadline falls at or before time } t_2\text{)} \end{aligned}$$

Thus, $\eta_i(t_1, t_2)$ is the number of times task τ_i must execute to completion on $[t_1, t_2)$ to meet all its deadlines on $(t_1, t_2]$.

Theorem 6.4 ([BHR '93]) *EDF produces a valid schedule for T, a task set with integer valued parameters, if and only if*

$$\begin{aligned} &1)\quad U(T) \leq 1 \qquad \textit{and} \\ &2)\quad \sum_{i=1}^{n} \eta_i(t_1, t_2) e_i \leq t_2 - t_1 \textit{ for all } 0 \leq t_1 < t_2 \leq r + 2P \end{aligned}$$

Proof: We first show that if EDF produces a valid schedule, then conditions 1 and 2 are true.

Clearly, if condition 1 fails, then the task set is not schedulable by Theorem 3.1. As well, if condition 2 fails, then there exists some t_1, t_2 such that $\sum_{i=1}^{n} \eta_i(t_1, t_2) e_i > t_2 - t_1$. Thus, the amount of execution required on $[t_1, t_2)$ is greater than the amount of time available. Hence, there must be a missed deadline. Thus, if conditions 1 or 2 fail to hold, there is no valid schedule of T. Therefore, if there exists a valid schedule of the task set, conditions 1 and 2 must be met. Since EDF is optimal for this type of task set, if there exists a valid schedule of the task set, then EDF also produces a valid schedule. Thus, if EDF produces a valid schedule for T, then conditions 1 and 2 hold.

We now show that if conditions 1 and 2 are true, then EDF produces a valid schedule.

Let g be the schedule of T produced by EDF. Suppose g is not valid and both conditions hold. By Theorem 6.3, we then know some deadline in $(0, r+2P]$ is not met. Let t_2 be such a deadline, and task τ_k be such that τ_k overflows at time t_2. Then let $t_1 \geq 0$ be the minimal value such that there is no idle time on $[t_1, t_2)$, and all execution of tasks on $[t_1, t_2)$ correspond to deadlines at or before t_2. Note that these conditions guarantee that $0 \leq t_1 < t_2$, since there can be no idle time on $[t_2 - d_k, t_2)$, and the only tasks executing on that interval must have deadlines at or before t_2 by definition of EDF. By the definition of t_1, we know that all execution on $[t_1, t_2)$ must correspond to a release at or after t_1. Since there is no idle time on $[t_1, t_2)$ and all execution corresponds to releases on that interval, $\sum_{i=1}^{n} \eta_i(t_1, t_2)e_i > t_2 - t_1$. Thus, we have a contradiction to condition 2. Hence, there is no such missed deadline t_2. $\square$

To prepare for the complexity analysis, [BHR '93] shows that $\eta_i(t_1, t_2)$ can be efficiently computed.

Lemma 6.3 ([BHR '93])

$$\eta_i(t_1, t_2) = \max\left\{0, \left\lfloor \frac{t_2 - r_i - d_i}{p_i} \right\rfloor - \max\left\{0, \left\lceil \frac{t_1 - r_i}{p_i} \right\rceil + 1\right\}\right\}$$

Proof: By definition of $\eta_i(t_1, t_2)$, we know $t_1 \leq r_i + kp_i$. Solving for k, we have $k \geq \frac{t_1 - r_i}{p_i}$. The minimal such k is exactly $\max\left\{0, \left\lceil \frac{t_1 - r_i}{p_i} \right\rceil\right\}$. Also from the definition of $\eta_i(t_1, t_2)$, we know $r_i + kp_i + d_i \leq t_2$. Solving again for k, we have $k \leq \frac{t_2 - r_i - d_i}{p_i}$. The maximal such k is then $\left\lceil \frac{t_2 - r_i - d_i}{p_i} \right\rceil$. Hence, the total number of k's satisfying the definition of $\eta_i(t_1, t_2)$ is exactly the difference between the maximal k and the minimal k, or zero if $\left\lceil \frac{t_2 - r_i - d_i}{p_i} \right\rceil - \max\left\{0, \left\lceil \frac{t_1 - r_i}{p_i} \right\rceil\right\} \leq 0$. $\square$

6.5 Complexity of feasibility tests

As mentioned above, if for all $i, d_i = p_i$, then comparing the utilization of the given task set to one is a polynomial (linear) time algorithm that determines feasibility. Without that restriction, the feasibility problem is co-*NP*-complete in the strong sense. Note that since EDF is optimal among scheduling algorithms for all tasks sets, this result then implies that the general question of schedulability of a given task set on a uniprocessor system is also co-*NP*-complete in the strong sense. We will follow the work of [LM '80] to reduce the Simultaneous Congruences Problem (SCP), which is shown to be *NP*-complete in the strong sense in [BHR '93], to determining if a task set is not feasible. Note that this reduction is very similar to the reduction found in Section 5.5.

First, we recall SCP: Given n ordered pairs of positive integers $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ and a positive integer $K, 2 \leq K \leq n$, is there a subset of $l \geq K$ ordered pairs $(a_{i_1}, b_{i_1}), (a_{i_2}, b_{i_2}), \ldots, (a_{i_l}, b_{i_l})$ such that there is a positive integer x such that $x \equiv a_{i_j} \bmod b_{i_j}$ for each $1 \leq j \leq l$?

Now, the reduction. Given an instance of SCP, $(a_1, b_1), \ldots, (a_n, b_n)$ and K, we construct the following task system, T, of $n+1$ tasks: for all $i, 1 \leq i \leq n, \tau_i = (1, K, Kb_i, Ka_i)$. $\tau_{n+1} = (1, K, K, 0)$. Since each task has a computation time of 1, a deadline span of K, and its release times, deadline spans and periods are multiples of K (thus releases only occur at time values that are multiples of K), then an overflow will occur if and only if $K+1$ (or more) tasks are released at a given timestamp that is a multiple of K. As τ_{n+1} is requested at every timestamp that is a multiple of K, then there is overflow if and only if K (or more) other tasks (namely, τ_1 through τ_n) release at any given timestamp that is a multiple of K. Given some time x, simple algebra dictates that a task τ_i has a release at time x if and only if $x \equiv Ka_i \bmod Kb_i$. Hence, there is overflow if and only if there is some positive integer x and $l \geq K$ tasks $\{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_l}\} \subseteq \{\tau_j\}_{j=1}^{n}$ such that $x \equiv Ka_{i_k} \bmod Kb_{i_k}$ for all $1 \leq k \leq l$. Therefore x is a multiple of K, and for $y = \frac{x}{K}$, $y \equiv a_{i_k} \bmod b_{i_k}$. Note that this condition is exactly the condition for a solution to SCP. This reduction is polynomial in time, so if there exists a polynomial time algorithm to determine if a task set is not schedulable on a uniprocessor system, then there exists a polynomial time algorithm to solve SCP. Since SCP is *NP*-complete in the strong sense, determining if a task set is **not** schedulable on a uniprocessor is also *NP*-complete in the strong sense. Thus, the feasibility problem (determining if a task set is schedulable on a uniprocessor) is co-*NP*-hard in the strong sense.

Now, we must show that the feasibility question is in co-*NP*. Using Theorem 6.4, we see that given a task set T for which EDF will not produce a valid schedule, either $U(T) > 1$ (which is computable in polynomial time), or

$$\sum_{i=1}^{n} \eta_i(t_1, t_2) e_i \leq t_2 - t_1 \text{ for all } 0 \leq t_1 < t_2 \leq r + 2P$$

fails to hold. Thus, if $U(T) \leq 1$ there is some t_1 and t_2 for which $\sum_{i=1}^{n} \eta_i(t_1, t_2) e_i > t_2 - t_1$. By nondeterministically choosing such t_1 and t_2, one may compute $\sum_{i=1}^{n} \eta_i(t_1, t_2)$ in polynomial time: There are n computations of the η_i's, each of which may be computed in $O(1)$ time (by Lemma 6.3). Thus, the feasibility question is in co-*NP* and is co-*NP*-hard in the strong sense. Hence, it is co-*NP*-complete in the strong sense.

7 Modified Least Laxity First

In covering EDF and a scheduling algorithm known as Least Laxity First (LLF) found in [Mo '83], we noticed that both shared a common structure in determining task priorities. Both used the next deadline of a given task and the current time in computing priorities. LLF also used the remaining amount of execution for the current release of the task. We noted that EDF and LLF could be seen as the same type of scheduling, by using a multiplicative factor on the remaining amount of execution – EDF using a factor of 0, and LLF using a factor of 1. This prompted us to question what would occur if that factor were something other than 0 or 1, and we discovered a resulting scheduling technique that was also optimal, but more general than either EDF, LLF, or any scheduling algorithm that was a hybrid of the two.

7.1 Definition

We define modified least laxity scheduling (MLLF) with a factor of f, $f \in \mathbb{R}$, as a dynamic priority scheduling algorithm. The priority of a given task τ_i at time t is exactly its modified laxity at time t,

$$ml_i(t) = d_i(t) - t - f \cdot e_i(t)$$

where $d_i(t)$ is the next deadline of τ_i after time t, and $e_i(t)$ is the amount of execution remaining for τ_i to complete this invocation. Formally,

$$d_i(t) = \begin{cases} r_i + d_i & : \quad t < r_i \\ r_i + \left\lfloor \frac{t-r_i}{p_i} \right\rfloor p_i + d_i & : \quad t \geq r_i \end{cases}$$

and

$$e_i(t) = \begin{cases} 0 & : \quad t < r_i \\ e_i - \int_{r_i + \left\lfloor \frac{t-r_i}{p_i} \right\rfloor p_i}^{t} \chi_{g,\tau_i}(x)\, dx & : \quad t \geq r_i \end{cases}$$

It should be noted that if $f = 0$,

$$ml_i(t) = d_i(t) - t$$

and MLLF is identical to EDF. Additionally, if $f = 1$,

$$ml_i(t) = d_i(t) - t - e_i(t)$$

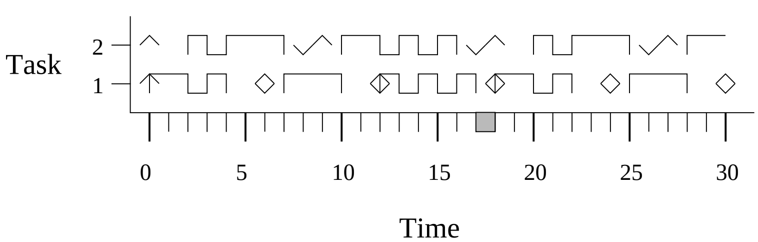
and MLLF is identical to LLF. Therefore MLLF is a general scheduling algorithm encompassing both EDF and LLF.

7.2 Example

As we mentioned above, MLLF is a generalization of EDF. Therefore, the example discussed for EDF is also a sample of MLLF with a laxity factor of 0.

We now consider the task set $T = \{\tau_i\}_{i=1}^{2}$, where $\tau_1 = (3, 6, 6, 0)$ and $\tau_2 = (4, 8, 9, 0)$. Under MLLF with a laxity factor of 1, the schedule is determined as follows (where the task with the lower modified laxity executes, and ties are broken arbitrarily):

0	$ml_1(0) = 6 - 0 - 1 \cdot 3 = 3$	$ml_2(0) = 8 - 0 - 1 \cdot 4 = 4$
1	$ml_1(1) = 6 - 1 - 1 \cdot 2 = 3$	$ml_2(1) = 8 - 1 - 1 \cdot 4 = 3$
2	$ml_1(2) = 6 - 2 - 1 \cdot 1 = 3$	$ml_2(2) = 8 - 2 - 1 \cdot 4 = 2$
3	$ml_1(3) = 6 - 3 - 1 \cdot 1 = 2$	$ml_2(3) = 8 - 3 - 1 \cdot 3 = 2$
4	τ_1 not active	$ml_2(4) = 8 - 4 - 1 \cdot 3 = 1$
5	τ_1 not active	$ml_2(5) = 8 - 5 - 1 \cdot 2 = 1$
6	$ml_1(6) = 12 - 6 - 1 \cdot 3 = 3$	$ml_2(6) = 8 - 6 - 1 \cdot 1 = 1$
7	$ml_1(7) = 12 - 7 - 1 \cdot 3 = 2$	τ_2 not active
8	$ml_1(8) = 12 - 8 - 1 \cdot 2 = 2$	τ_2 not active
9	τ_1 not active	$ml_2(9) = 17 - 9 - 1 \cdot 4 = 4$
10	τ_1 not active	$ml_2(10) = 17 - 10 - 1 \cdot 3 = 4$
11	τ_1 not active	$ml_2(11) = 17 - 11 - 1 \cdot 2 = 4$
12	$ml_1(12) = 18 - 12 - 1 \cdot 3 = 3$	$ml_2(12) = 17 - 12 - 1 \cdot 1 = 4$
etc.		

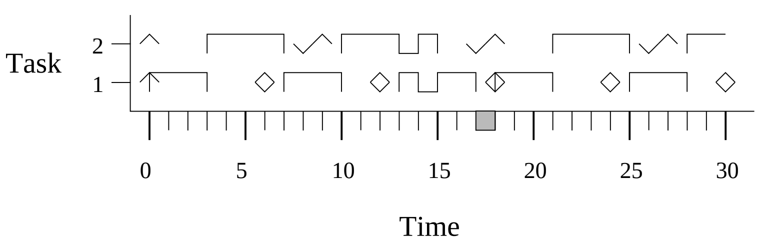


And compare those results to a schedule using MLLF with a factor of $\frac{1}{2}$:

0	$ml_1(0) = 6 - 0 - \frac{1}{2} \cdot 3 = 4\frac{1}{2}$	$ml_2(0) = 8 - 0 - \dfrac{1}{2} \cdot 4 = 6$
1	$ml_1(1) = 6 - 1 - \frac{1}{2} \cdot 2 = 4$	$ml_2(1) = 8 - 1 - \dfrac{1}{2} \cdot 4 = 5$

2	$ml_1(2) = 6 - 2 - \frac{1}{2} \cdot 1 = 3\frac{1}{2}$	$ml_2(2) = 8 - 2 - \frac{1}{2} \cdot 4 = 4$
3	τ_1 not active	$ml_2(3) = 8 - 3 - \frac{1}{2} \cdot 4 = 3$
4	τ_1 not active	$ml_2(4) = 8 - 4 - \frac{1}{2} \cdot 3 = 2\frac{1}{2}$
5	τ_1 not active	$ml_2(5) = 8 - 5 - \frac{1}{2} \cdot 2 = 2$
6	$ml_1(6) = 12 - 6 - \frac{1}{2} \cdot 3 = 4\frac{1}{2}$	$ml_2(6) = 8 - 6 - \frac{1}{2} \cdot 1 = 1\frac{1}{2}$
7	$ml_1(7) = 12 - 7 - \frac{1}{2} \cdot 3 = 3\frac{1}{2}$	τ_2 not active
8	$ml_1(8) = 12 - 8 - \frac{1}{2} \cdot 2 = 3$	τ_2 not active
9	$ml_1(7) = 12 - 7 - \frac{1}{2} \cdot 1 = 2\frac{1}{2}$	$ml_2(9) = 17 - 9 - \frac{1}{2} \cdot 4 = 6$
10	τ_1 not active	$ml_2(10) = 17 - 10 - \frac{1}{2} \cdot 4 = 5$
11	τ_1 not active	$ml_2(11) = 17 - 11 - \frac{1}{2} \cdot 3 = 4\frac{1}{2}$
12	$ml_1(12) = 18 - 12 - \frac{1}{2} \cdot 3 = 4\frac{1}{2}$	$ml_2(12) = 17 - 12 - \frac{1}{2} \cdot 1 = 4$

etc.



7.3 MLLF as an optimal scheduler

Our goal in this section is to show that if a given task set has a valid schedule, then MLLF with $0 \leq f \leq 1$ will also produce a valid schedule. After proving this result, we will show that the restrictions on f are necessary for optimality. Prior to the main theorem, we will first prove a preliminary lemma regarding the change of laxity factors over time.

Lemma 7.1 *Given a task set T, a task $\tau_i \in T$, and a schedule g of T created by MLLF, If*

$d_i(t) \neq t+1$ and τ_i is active at time t, then

$$ml_i(t+1) = \begin{cases} ml_i(t) - 1 & : \quad g(t) \neq \tau_i \\ ml_i(t) - 1 + f & : \quad g(t) = \tau_i \end{cases}$$

Proof: By definition of MLLF,

$$\begin{aligned} ml_i(t+1) &= d_i(t+1) - (t+1) - fe_i(t+1) \\ &= d_i(t+1) - t - fe_i(t+1) - 1 \end{aligned} \tag{34}$$

Since $d_i(t) \neq t+1$, we know $d_i(t) = d_i(t+1)$. If $g(t) \neq \tau_i$, then $e_i(t+1) = e_i(t)$. Thus, equation (34) becomes

$$ml_i(t) = d_i(t) - t - fe_i(t) - 1$$

which proves the first part of the lemma. If $g(t) = \tau_i$, then $e_i(t) = e_i(t+1) + 1$. Thus, equation (34) becomes

$$\begin{aligned} ml_i(t+1) &= d_i(t) - t - f(e_i(t) - 1) - 1 \\ &= d_i(t) - t - fe_i(t) + f - 1 \end{aligned}$$

which proves the second part of the lemma. □

7.3.1 Necessary conditions for optimality

With MLLF, we make no assumptions about synchronicity. We assume that for any given task τ_i, $d_i \leq p_i$. MLLF will be proven optimal where $0 \leq f \leq 1$.

7.3.2 Proof of optimality

We will do most of the work of this section in the following theorem. This theorem provides all the tools we need to use induction to show that MLLF with $0 \leq f \leq 1$ is optimal.

Theorem 7.1 *Let T be a task set of n tasks, and g be a valid schedule of T. Let $t \in \mathbb{Z}_+$, and $0 \leq f \leq 1$. Then there exists a valid schedule h of T such that for all $u \in \mathbb{Z}_+$ such that $u < t$, $h(u) = g(u)$; and h schedules by MLLF with a laxity factor of f at time t.*

To prove this theorem, we will construct h past time t, and prove that h is a valid schedule of T.

Proof: We divide our considerations according to the tasks $g(t)$ and $h(t)$.

Case 1: $g(t) = h(t)$. We then define $h = g$. Thus, h schedules at time t by MLLF, is identical to g on the interval $[0, t)$, and is a valid schedule of T.

Case 2: $g(t) \neq h(t)$. Let τ_i be the task such that $g(t) = \tau_i$. Let τ_j be the task such that $h(t) = \tau_j$. Note that then both τ_i and τ_j must be active at time t in *both* g and h, since $g = h$ for all $u \in [0, t)$, and neither τ_i nor τ_j have satisfied their releases prior to time t. Recall that τ_i's first deadline past a given time u is denoted $d_i(u)$, and τ_j's first deadline past u is denoted $d_j(u)$.

Subcase 2.A: If there exists some time v such that $t < v < \min(d_j(t), d_i(t))$ where $g(v) = \tau_j$, define

$$h(u) = \begin{cases} g(u) & : \quad \forall u \notin \{t, v\} \\ \tau_i & : \quad u = v \\ \tau_j & : \quad u = t \end{cases}$$

Note that h is identical to g except at times t and v. Since

$$\begin{array}{ll} g(t) = \tau_i & \qquad h(t) = \tau_j \\ g(v) = \tau_j & \qquad h(v) = \tau_i \end{array}$$

we know that all tasks other than τ_i and τ_j are scheduled in h exactly as they are in g. In fact, τ_i and τ_j are scheduled in h exactly as they are in g with the exception of the executions corresponding to their deadlines at $d_i(t)$ and $d_j(t)$. Since all deadlines are met in g, then we know all deadlines other than $d_i(t)$ for τ_i and $d_j(t)$ for τ_j are met in h. Thus, to show that h is valid, we merely must show that those deadlines are met in h. Thus, we must prove that

$$\sum_{u=t}^{d_i(t)-1} \chi_{h,\tau_i}(u) = e_i(t)$$

and

$$\sum_{u=t}^{d_j(t)-1} \chi_{h,\tau_j}(u) = e_j(t)$$

We will prove the result for τ_i; the proof for τ_j is identical with the exception of the subscript.

Since τ_i meets its deadline at $d_i(t)$ in g, then

$$\begin{aligned}
\sum_{u=t}^{d_i(t)-1} \chi_{g,\tau_i}(u) &= e_i(t) \\
\sum_{u=t}^{d_i(t)-1} \chi_{g,\tau_i}(u) - 1 - 0 + 0 + 1 &= e_i(t) \\
\sum_{u=t}^{d_i(t)-1} \chi_{g,\tau_i}(u) - \chi_{g,\tau_i}(t) - \chi_{g,\tau_i}(v) + \chi_{h,\tau_i}(t) + \chi_{h,\tau_i}(v) &= e_i(t)
\end{aligned}$$

Since $g(u) = h(u)$ for all $u \in [t, d_i(t))$ where $u \neq t$ and $u \neq v$, then we have the desired result, namely

$$\sum_{u=t}^{d_i(t)-1} \chi_{h,\tau_i}(u) = e_i(t)$$

Note that for this proof to work, it is required that $v \in [t, d_i(t))$.

For the same reasons, τ_j meets its deadline at $d_j(t)$ in h. Hence, h is a valid discrete schedule of T. By definition of h, h is identical to g on $[0, t)$, and schedules by MLLF at time t.

Subcase 2.B: The last case to consider is when there is no such time v such that $t < v < \min(d_j(t), d_i(t))$ with $g(v) = \tau_j$. By contradiction, we will show that this subcase can never hold. To do so, we will focus on the the modified laxities of τ_i and τ_j at times t and $d_i(t) - 1$.

In g, τ_j is active at time t and τ_j meets its deadline at $d_j(t)$, so τ_j must be scheduled in g for at least one time unit between t and $d_j(t)$. By the subcase assumption, τ_j is not scheduled in g on $[t, \min(d_j(t), d_i(t)))$. For τ_j to meet its deadline at $d_j(t)$, we must have $d_i(t) < d_j(t)$ – otherwise τ_j is active at t, and is not scheduled before its corresponding deadline. By the same logic, there must exist $e_j(t)$ time units on $[d_i(t), d_j(t))$ where τ_j is scheduled. Therefore, $1 \leq e_j(t) \leq d_j(t) - d_i(t)$. Now we compare the modified laxities of τ_i and τ_j at time t. First, we consider τ_i:

$$ml_i(t) = d_i(t) - t - fe_i(t)$$

Since $f \geq 0$ and $e_i(t) > 0$,

$$ml_i(t) \leq d_i(t) - t \tag{35}$$

with equality if and only if $f = 0$. Now, for τ_j we have the following:

$$ml_j(t) = d_j(t) - t - fe_j(t)$$

Since $0 \leq f \leq 1$ and $0 < e_j(t) \leq d_j(t) - d_i(t)$,

$$\begin{aligned} ml_j(t) &\geq d_j(t) - t - e_j(t) \\ &\geq d_j(t) - t - (d_j(t) - d_i(t)) \\ &= d_i(t) - t \end{aligned} \tag{36}$$

with equality if and only if $f = 1$ (and $e_j(t) = d_j(t) - d_i(t)$). Combining equations (35) and (36) along with the knowledge that f cannot be both 0 and 1 at the same time, we have $ml_i(t) < ml_j(t)$. Therefore, at time t, task τ_i has a lower modified laxity than task τ_j. However, this contradicts the case 2 assumption that $h(t) = \tau_j$, since h schedules by MLLF at time t. Therefore, there must be some time v with $t < v < \min(d_j(t), d_i(t))$ with $g(v) = \tau_j$.

We have thus shown that we may produce an h as dictated by the theorem in all possible cases.

Thus, if g is valid, then there is a schedule identical to g on $[0, t)$, that schedules by MLLF at time t, and is valid. $\square$

7.3.3 MLLF as an optimal scheduler

Since MLLF is a generalization of EDF, it should follow that MLLF, like EDF, is optimal. However, there are some restrictions that must be applied to ensure MLLF is optimal. The laxity factor must be between zero and one (inclusive), and the task sets must have integer parameters.

Theorem 7.2 *MLLF with a varying laxity factor between zero and one (inclusive) is an optimal scheduling algorithm for task sets with integer valued parameters.*

Proof: We prove this theorem by induction. Let T be a task set, and let g be a valid schedule of T. Let f_0 be such that $0 \leq f_0 \leq 1$. Then by Theorem 7.1 applied to time 0, we know that there is a valid schedule of T that schedules by MLLF at time 0.

Now let $t > 0$. For all u such that $0 \leq u < t$, let f_u be such that $0 \leq f_u \leq 1$. Our inductive assumption is that there is a valid schedule h of T such that for each u in $[0, t)$, h schedules by MLLF with the factor f_u at the time u. Thus, h is a valid schedule of T. Let f_t be such that $0 \leq f_t \leq 1$. By Theorem 7.1, we know that there is a valid schedule of T, identical to h on $[0, t)$ which schedules at time t by MLLF with the factor f_t.

By induction, we have shown that for any task set with a valid schedule, MLLF with varying laxity factors (between zero and one inclusive) produces a valid schedule.

That is to say, the result shows that given a function $z : \mathbb{Z}_+ \mapsto [0, 1]$, and some valid discrete schedule g, the schedule h produced by using MLLF with factor $z(t)$ at time t for all $t \geq 0$ is valid. Therefore, MLLF with varying laxity factors is optimal since it produces a valid schedule for any task set that has a valid schedule.

A direct result of this theorem is that MLLF with a fixed laxity factor (between zero and one inclusive) is optimal. $\square$

We now will prove that the limitations on the laxity factor are strict. That is to say, for $f < 0$ or $f > 1$, there exists a task set with utilization equal to one such that MLLF with a laxity factor of f yields an invalid schedule.

We now proceed to prove that the laxity factor must be at least zero for MLLF to be optimal. Given a laxity factor less than zero, we will produce a task set with a utilization of one, yet that MLLF with that laxity factor will not yield a valid schedule.

Theorem 7.3 *MLLF is not optimal for fixed laxity factors less than zero.*

Our proof obligation here is merely to show that given a laxity factor less than zero, there is a task set that has a valid schedule such that MLLF with the given laxity factor does not produce a valid schedule of that task set.

Proof: Let $f < 0$. Then there exists some $n > 3$ such that $f \leq -\frac{1}{n}$. Let T be the task set of two tasks such that $\tau_1 = (3n+1, 18n^2+6n, 18n^2+6n, 0)$ and $\tau_2 = (36n^2-6n, 36n^2, 36n^2, 0)$.

First, we show that $U(T) = 1$ (therefore by Theorems 6.1 and 7.2, T is schedulable by EDF and by MLLF with a laxity factor between 0 and 1).

$$\begin{aligned} U(T) &= \frac{e_1}{p_1} + \frac{e_2}{p_2} \\ &= \frac{3n+1}{18n^2+6n} + \frac{36n^2-6n}{36n^2} \end{aligned}$$

$$\begin{aligned}
&= \frac{(3n+1)(36n^2)}{(18n^2+6n)(36n^2)} + \frac{(36n^2-6n)(18n^2+6n)}{(18n^2+6n)(36n^2)} \\
&= \frac{(108n^3+36n^2)+(648n^4+108n^3-36n^2)}{648n^4+216n^3} \\
&= \frac{648n^4+216n^3}{648n^4+216n^3} \\
&= 1
\end{aligned}$$

Next, we show that if τ_1 meets its deadline at $p_1 = 18n^2+6n$, then τ_2 will overflow at time $p_2 = 36n^2$. If τ_1 meets its deadline at p_1, then τ_1 executed for $3n+1$ time units on $(0, 18n^2+6n)$. Therefore, τ_2 executed for $18n^2+6n-(3n+1)$ time units on the same interval. Thus, at time $18n^2+6n$, τ_2 has $(36n^2-6n)-(18n^2+6n-(3n+1))$ time units left to execute. Thus, $e_2(18n^2+6n) = 18n^2-9n+1$. Now, let us discern which task has priority at time $18n^2+6n$: For τ_1,

$$\begin{aligned}
ml_1(18n^2+6n) &= d_1(18n^2+6n)-(18n^2+6n)-fe_1(18n^2+6n) \\
&= 36n^2+12n-18n^2-6n-f(3n+1) \\
&= 18n^2+6n-f(3n+1) \qquad (37)
\end{aligned}$$

For τ_2,

$$\begin{aligned}
ml_2(18n^2+6n) &= d_2(18n^2+6n)-(18n^2+6n)-fe_2(18n^2+6n) \\
&= 36n^2-18n^2-6n-f(18n^2-9n+1) \\
&= 18n^2-6n-f(18n^2-9n+1) \qquad (38)
\end{aligned}$$

So, we must compare $18n^2+6n-f(3n+1)$ and $18n^2-6n-f(18n^2-9n+1)$. By assumption on n, we know $n \geq 3$.

$$\begin{aligned}
n &> 2 \\
6n &> 12 \\
18n-12 &> 12n \\
\frac{1}{n}(18n^2-12n) &> 12n
\end{aligned}$$

Since $f \leq -\frac{1}{n}$, then $-f \geq \frac{1}{n}$. Hence,

$$\begin{aligned}
-f(18n^2-12n) &> 12n \\
f(12n-18n^2) &> 12n \\
f(3n+1-18n^2+9n-1) &> 12n
\end{aligned}$$

$$\begin{aligned} f(3n+1) - f(18n^2 - 9n + 1) &> 12n \\ -6n - f(18n^2 - 9n + 1) &> 6n - f(3n+1) \\ 18n^2 - 6n - f(18n^2 - 9n + 1) &> 18n^2 + 6n - f(3n+1) \end{aligned} \tag{39}$$

Combining equations (37), (38), and (39), we have

$$ml_2(18n^2 + 6n) > ml_1(18n^2 + 6n)$$

Additionally, by Lemma 7.1, we know that the modified laxity of the task on the processor changes by $(-1 + f)$ each time unit. The modified laxity of any task not on the processor changes by -1 every time unit. Since $f < 0$, then the modified laxity of the task on the processor will decrease by more than that of the non-scheduled task every time unit. Thus, once a task is on the processor, it can be pre-empted only if another task is released. Since τ_1 is on the processor at time $18n^2 + 6n$, then it will execute to completion (there are no releases until time $36n^2$). Thus, τ_1 is on the processor for $3n + 1$ time units on the interval $(18n^2 + 6n, 36n^2)$. Note, however, that at time $18n^2 + 6n$, task τ_2 has $18n^2 - 9n + 1$ time units of execution remaining, and there are $18n^2 - 6n$ time units until τ_2's deadline. Since τ_1 is scheduled for $3n+1$ of those time units, then there are $18n^2 - 6n - (3n+1)$ time units for τ_2 to execute. Therefore the amount of time available, $18n^2 - 9n - 1$ is less than the amount of execution remaining, $18n^2 - 9n + 1$ for τ_2. τ_2 will therefore miss its deadline at $36n^2$. $\square$

We now proceed to prove that the laxity factor must be at most one for MLLF to be optimal. Given a laxity factor greater than one, we will produce a task set with a utilization of one, yet that MLLF with that laxity factor will not yield a valid schedule.

Theorem 7.4 *MLLF is not optimal for fixed laxity factors greater than one.*

Our proof obligation here is merely to show that given a laxity factor greater than one, there is a task set that has a valid schedule such that MLLF with the given laxity factor does not produce a valid schedule of that task set.

Proof: Let $f > 1$. Then there exists some $n > 0$ such that $f \geq \frac{n+1}{n}$. Let T be the task set of two tasks, $\tau_1 = (1, n+2, n+2, 0)$ and $\tau_2 = ((n+3)(n+1), (n+3)(n+2), (n+3)(n+2), 0)$.

First, we show that $U(T) = 1$ (therefore by Theorems 6.1 and 7.2, T is schedulable by EDF and MLLF with a laxity factor between 0 and 1).

$$U(T) = \frac{e_1}{p_1} + \frac{e_2}{p_2}$$

$$
\begin{aligned}
&= \frac{1}{n+2} + \frac{(n+3)(n+1)}{(n+3)(n+2)} \\
&= \frac{(n+3)+(n+3)(n+1)}{(n+3)(n+2)} \\
&= \frac{(n+3)(n+2)}{(n+3)(n+2)} \\
&= 1
\end{aligned}
$$

Next, we show that τ_1 will not be scheduled at any time on the interval $[0, n+1]$. If it isn't scheduled at any of those times, then we know it cannot meet its deadline at $n+2$.

Consider that for $t \leq n+1$, if τ_1 is not scheduled before time t, then

$$
\begin{aligned}
ml_1(t) &= d_1(t) - t - fe_1(t) \\
&= (n+2) - t - f \cdot 1 \\
&= n + 2 - t - f \cdot 1 \qquad (40)
\end{aligned}
$$

For $t \leq n+1$, if τ_2 is scheduled on the entire interval $[0, t)$, then

$$
\begin{aligned}
ml_2(t) &= d_2(t) - t - fe_2(t) \\
&= (n+3)(n+2) - t - f((n+3)(n+1) - t) \qquad (41)
\end{aligned}
$$

So now we wish to show that for $t \in [0, n+1]$, $(n+3)(n+2) - t - f((n+3)(n+1) - t) < n + 2 - t - f$. By showing this equation to be true, then (by induction), we know that τ_1 is not scheduled at any time on the interval $[0, n+1]$.

$$
\begin{aligned}
0 &< 1 \\
n^3 + 4n^2 + 4n &< n^3 + 4n^2 + 4n + 1 \\
n^3 + 4n^2 + 4n &< n^3 + 3n^2 + n + n^2 + 3n + 1 \\
n^3 + 4n^2 + 4n &< (n+1)(n^2 + 3n + 1) \\
n^3 + 4n^2 + 4n &< (n+1)(n^2 + 3n + 2 - 1) \\
n(n+2)(n+2) &< (n+1)((n+2)(n+1) - 1) \\
(n+2)(n+2) &< \frac{n+1}{n}((n+2)(n+1) - 1) \\
(n+2)(n+2) &< \frac{n+1}{n}((n+3)(n+1) - (n+1) - 1)
\end{aligned}
$$

Since $t \leq n+1$,

$$\begin{aligned}(n+2)(n+2) &< \frac{n+1}{n}((n+3)(n+1)-t-1) \\ (n+2)(n+2) &< f((n+3)(n+1)-t-1) \\ (n+2)(n+2)-f((n+3)(n+1)-t) &< -f \\ (n+3)(n+2)-t-f((n+3)(n+1)-t) &< (n+2)-t-f\end{aligned}$$

Therefore, by equations (40) and (41),

$$ml_2(t) < ml_1(t) \tag{42}$$

We now show (by induction) that τ_2 is scheduled on the entire interval $[0, n+2)$. Equation (42) is true for $t = 0$, so τ_2 is scheduled at time 0. Let t be such that $0 < t < n+1$. Now we assume that τ_2 is scheduled on the entire interval $[0, t)$. Thus equation (42) holds for time t, and therefore $ml_2(t) < ml_1(t)$. Thus, τ_1 is not scheduled at any time on the interval $[0, n+1]$, and it therefore misses its deadline at $n+2$. □

It is interesting to note that [Mo '83] remarks that, "There are in fact an infinite number of totally on-line optimal schedulers, e.g., any combination of the earliest deadline first and the least slack algorithm may conceivably be used in a run-time scheduler to minimize process switching overheads." In essence, MLLF with a variable laxity factor extends that remark – since the remark in [Mo '83] is merely a restriction of the above function z (to the range $\{0, 1\}$). In fact, our result is strictly more general in the types of allowable schedules (that is to say, EDF and LLF swapping cannot produce all schedules that variable laxity factors can produce). Consider the task set $\{\tau_1 = (2, 16, 16, 0), \tau_2 = (6, 17, 17, 0), \tau_3 = (10, 20, 20, 0)\}$, a synchronous task set with a utilization approximately equal to .978 (which is less than 1, so the task set is schedulable with any of the algorithms under discussion). At time 0, EDF will discern that the nearest deadline is that of task τ_1, hence EDF would schedule τ_1 at time 0. At time 0, LLF (MLLF with a factor of 1) determines that the laxity of τ_1 is 14, the laxity of τ_2 is 11, and the laxity of τ_3 is 10. Hence, LLF would schedule τ_3 at time 0. At time 0, MLLF with a factor of $\frac{1}{2}$ will determine the modified laxities of the tasks are 15, 14, and 15 (respectively). Thus, MLLF with a factor of $\frac{1}{2}$ will schedule τ_2 at time 0. Since neither EDF nor LLF schedules τ_2 at time 0, we have a valid schedule under MLLF that cannot be produced with EDF/LLF swapping. Therefore, MLLF with a variable laxity factor is strictly more general than EDF and LLF swapping.

Another note of interest regarding MLLF is that if one is producing a non-discrete schedule, then MLLF is probably an unwise choice (unless one uses a laxity factor of 0 to produce EDF). The reason is that when two (or more) tasks have identical laxity, if the processor schedules one, it must then swap back and forth between the two until one has completed execution. The number of task swaps will be quite high, and usually the cost associated with swapping tasks is non-trivial. Specifically, if tasks τ_1 and τ_2 have identical modified laxities at time t, then the algorithm may select either to schedule. Without loss of generality, let us assume that τ_1 is then scheduled for ϵ time units. Consider that for $f > 0$:

$$\begin{aligned}
ml_1(t) &= d_1(t) - t - fe_1(t) \\
&\quad \tau_1 \text{ is then scheduled for } \epsilon \text{ time units:} \\
ml_1(t+\epsilon) &= d_1(t) - (t+\epsilon) - f(e_1(t) - \epsilon) \\
ml_1(t+\epsilon) &= d_1(t) - t - f(e_1(t)) - \epsilon + f\epsilon \\
ml_1(t+\epsilon) &= ml_1(t) - \epsilon + f\epsilon
\end{aligned}$$

And

$$\begin{aligned}
ml_2(t) &= d_2(t) - t - fe_2(t) \\
&\quad \tau_2 \text{ is not scheduled for } \epsilon \text{ time units:} \\
ml_2(t+\epsilon) &= d_2(t) - (t+\epsilon) - fe_2(t) \\
ml_2(t+\epsilon) &= d_2(t) - t - fe_2(t) - \epsilon \\
ml_2(t+\epsilon) &= ml_2(t) - \epsilon \\
&\quad \text{since } ml_1(t) = ml_2(t) \\
ml_2(t+\epsilon) &= ml_1(t) - \epsilon
\end{aligned}$$

Thus,

$$ml_1(t+\epsilon) > ml_2(t+\epsilon)$$

and so at time $t + \epsilon$, task τ_2 will be scheduled. Note that (by similar computations) after ϵ further time units, τ_1 and τ_2 will again have identical modified laxities, and the swapping process will begin again. Note that the only laxity factor that can avoid this swapping is 0 – when one schedules with EDF. Clearly, as ϵ tends to 0, the amount of swapping becomes infinite. If for no other reason, this explanation provides the motivation to use MLLF solely for discrete scheduling.

7.4 Complexity of feasibility tests

As we have already shown in Section 6.5, the feasibility problem for a task set without resources is co-*NP*-complete. Since both EDF and MLLF are optimal, then any feasibility algorithm for one will also determine feasibility for the other. Hence, if $d_i = p_i$ for all tasks τ_i, then MLLF produces a valid schedule if and only if $U \leq 1$. Additionally, a sufficient test for schedulability is

$$\sum_{i=1}^{n} \frac{e_i}{d_i} \leq 1$$

As explained in Section 6.5, the above test is not necessary for schedulability.

The general feasibility test, as shown in Section 6.5, is co-*NP*-complete in the strong sense.

8 Conclusions

We have seen that all four scheduling algorithms have their drawbacks – namely, from the development in our work on EDF, we know that the general question of schedulability for a task set is co-*NP*-complete in the strong sense. However, this does not rule out the possiblity that a given task set will lend itself to a less demanding feasibility test. For example, we know that for synchrounous task sets where deadline spans are identical to periods, that any task set of n tasks has a valid schedule under RM if the utilization of that task set is at most $n(2^{\frac{1}{n}} - 1)$ – and therefore the task set also has a valid schedule under DM. Additionally, if the utilization is at most 1, we know that EDF will produce a valid schedule for the task set – and therefore the task set also has a valid schedule under MLLF with any laxity factor between zero and one (inclusive).

The difficulty arises for task sets where deadline spans are not identical to periods. In these cases, the utilization of a task set may have very little to do with its schedulablity. For example, for $n \in \mathbb{Z}^+$, the task set $\{(1, 1, n, 0), (1, 1, n, 0)\}$ has no valid schedule. Since this holds for any $n > 0$, we see that a task set may have an extremely small utilization, and still have no valid schedule. In these cases, feasibility tests appear to become quite intractible for large task sets since the general question of feasibility is co-*NP*-complete in the strong sense.

Some open questions remain, however, whose answers may paint a brighter picture on the feasibility question. We do have a pseudo-polynomial time test for feasibility under RM for synchronous task sets where deadline spans are identical to periods. We know that in that case the feasibility question for RM is in *NP*, but have no results stating whether the question is *NP*-complete. We do not know if there is an optimal static priority scheduling algorithm for asynchronous task sets. We also have provided a new scheduling algorithm

(MLLF) that generalizes the two optimal dynamic priority scheduling algorithms one sees in the literature. Perhaps this unification will provide new light in which to consider dynamic priority scheduling, and may lead to discerning new classes of task sets that have polynomial time feasibility tests. However, MLLF was shown to be optimal when considering discrete schedules – this is also how LLF (see [Mo '83]) is considered – but was not developed for schedules over continuous time.

Overall, we have tried to provide clarity to some of the major scheduling algorithms in the field, and to show their relationships. There are many issues to consider in hard-real-time scheduling, and hopefully this paper has provided solid groundwork for the algorithms we've covered.

9 Glossary

Notation: A periodic task $\tau_i = (e_i, d_i, p_i, r_i)$ is said to have an execution time of e_i, a deadline span of d_i, a period of p_i, and an initial release time of r_i. We concern ourselves solely with tasks where $e_i \leq d_i \leq p_i$.

$\mathbb{R}_+ = \{x \in \mathbb{R} \wedge x \geq 0\}$

$\mathbb{Z}_+ = \{x \in \mathbb{Z} \wedge x \geq 0\}$

$$\chi_{f,b}(a) = \begin{cases} 0 & : \quad f(a) \neq b \\ 1 & : \quad f(a) = b \end{cases}$$

Active: A task τ_i is active at time t if and only if there exists $k \in \mathbb{Z}_+$ such that $r_{i,k} \leq t < r_{i,k} + d_i$ and $\int_{x=r_{i,k}}^{t} \chi_{g,\tau_i}(x)\, dx < e$.

Critical instant: A task has a critical instant (under a given scheduling algorithm) at any release that yields the longest possible response time of that task for the specified scheduling algorithm and task set.

Deadline: τ_i is said to have deadlines at $d_{i,k}$, $k \in \mathbb{Z}_+$, where $d_{i,k+1} = r_i + kp_i + d_i$. $d_i(t)$ is the deadline of τ_i after time t, formally

$$d_i(t) = \begin{cases} r_i + d_i & : \quad t < r_i \\ r_i + \left\lfloor \frac{t-r_i}{p_i} \right\rfloor p_i + d_i & : \quad t \geq r_i \end{cases}$$

Discrete Schedule: A function $g : \mathbb{Z}_+ \mapsto T$

Execution: τ_i is said to have an execution time of e_i. Given a schedule g, $e_{i,g(t)}$ is the amount of execution remaining for the invocation of τ_i at time t. When clear, the g subscript will be omitted. Formally,

$$e_i(t) = \begin{cases} 0 & : \quad t < r_i \\ e_i - \int_{r_i + \left\lfloor \frac{t-r_i}{p_i} \right\rfloor p_i}^{t} \chi_{g,\tau_i}(x)\, dx & : \quad t \geq r_i \end{cases}$$

e_g, i, t is the amount of execution completed for the invocation of τ_i at time t. Formally,

$$e_{g,i,t} = \begin{cases} \int_R^t \chi_{g,\tau_i}(x) dx & : \quad t \geq r_i \\ e_i & : \quad t < r_i \end{cases}$$

where $R = \max_{j \in \mathbb{Z}_+}\{r_i + jp_i \leq t\}$. Thus, $e_{i,g(t)} + e_{g,i,t} = e_i$ for all i, g, and $t \geq 0$.

Fully utilized: A task set fully utilizes the processor under a given scheduling algorithm if that algorithm yields a valid schedule for the task set, but that algorithm fails to yield a valid schedule if any task's execution time is increased.

Meeting a deadline: τ_i meets its deadline at $d_{i,k}$ if there exists $l \in \mathbb{Z}_+$ such that $\int_{x=r_{i,k}}^{t} \chi_{g,\tau_i}(x)\, dx \geq e$

Optimality: Under given constraints, a scheduling algorithm is optimal if it produces a valid schedule for every task set that has a valid schedule under the same constraints.

Overflow: A task τ_i overflows or misses its deadline at $d_{i,k}$ if there exists $l \in \mathbb{Z}_+$ such that $d_{i,k} = r_{i,l} + d_i$ and $\int_{r_{i,k}}^{d_{i,k}} \chi_{g,\tau_i}(x)\, dx < e$.

Periodic task without resources: $\tau_i = (e_i, d_i, p_i, r_i)$

Prioritizing: Each task τ_i is assigned a corresponding number, P_i. P_i is τ_i's priority. Priorities may be either *static* (constant over time) or *dynamic* (change over time). Lower priority numbers correspond to higher priorities.

Priority based scheduling algorithm: An algorithm that assigns priorities to the tasks, and produces the following schedule: $g_P(t) = \tau_i$ such that τ_i is active at time t, and $\forall j \neq i$, $P_j < P_i \Rightarrow P_j$ is not active. If there are no active tasks at time t, then $g_P(t) = \emptyset$. Note that if two (or more) active process have the same priority, ties may be broken arbitrarily.

Release (Release time): A task τ_i is said to release (or have release times) at $r_{i,k+1}$, $k \in \mathbb{Z}_+$, where $r_{i,k+1} = r_i + kp_i$. We use the shift of one unit on k so that the first release, at time r_i, corresponds to $r_{i,1}$ (instead of $r_{i,0}$).

Response time: The response time of the k^{th} release of a task τ_i is the amount of time required to for τ_i to execute to completion (for that release). Technically, the response time for a schedule g of task τ_i's k^{th} release is

$$\min_{\left\{t : \int_{r_{i,k}}^{t} \chi_{g,\tau_i}(x)\, dx = e\right\}} t - r_{i,k}$$

Schedule: A function $g : \mathbb{R}_+ \mapsto T$

Satisfied release: Task τ_i's release at $r_{i,k}$ is satisfied if the given schedule meets τ_i's deadline $d_{i,k} = r_{i,k} + d_i$.

Task set T: A set of tasks, $\{\tau_i\}_{i=1}^n$, such that each task has a corresponding execution time (e_i), and a period (p_i).

Utilization: $U : T \mapsto \mathbb{R}_+$ is defined by

$$U(T) = \sum_{i=1}^{n} \frac{e_i}{p_i}$$

Valid: A valid schedule is one where all deadlines are met.

References

[ARJ '97] J. Anderson, S. Ramamurthy, and K. Jeffay, **Real-Time Computing with Lock-Free Shared Objects (Technical Report)**, scheduled to appear in *ACM Transactions on Computer Systems*, 15(2), 1997.

[ABD '95] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell and Andy J. Wellings, **Fixed Priority Pre-emptive Scheduling: An Historical Perspective**, *Real-Time Systems*, 8: 173-198, 1995.

[BHR '93] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier, **Feasibility problems for recurring tasks on one processor**, *Theoretical Computer Science*, 118: 3-20, 1993.

[BRH '90] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell, **Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor**, *Real-Time Systems*, 2: 301-324, 1990.

[CL '90] Min-ih Chen and Kwei-Jay Lin, **Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems**, *Real-Time Systems*, 2: 325-346, 1990.

[Je '92] Kevin Jeffay, **Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems**, *IEEE Real-Time Systems Symposium*, ,1992.

[La '74] Jacques Labetoulle, **Some Theorems on Real Time Scheduling**, appearing in: E. Gelenbe and R. Mahl, eds., *Computer Architectures and Networks*, (North-Holland Publishing, Amsterdam) 285-293, 1974.

[LSD '89] John Lehoczky, Lui Sha, and Ye Ding, **The Rate Monotonic Scheduling Algorithm: Exact Case Characterization And Average Case Behavior**, *Proc. IEEE Real-Time Systems Symposium* 166-171, 1989.

[LW '82] Joseph Y.-T. Leung and Jennifer Whitehead, **On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks**, *Performance Evaluation*, 2: 237-250, 1982.

[LM '80] Joseph Y.-T. Leung and M.L. Merrill, **A Note on Preemptive Scheduling of Periodic, Real-Time Tasks**, *Information Processing Letters*, 11(3): 115-118, 1980.

[LL '73] C.L. Liu and James W. Layland, **Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment**, *JACM*, 20(1): 174-189, 1973.

[Mo '83] Aloysius K. Mok, **Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment**, *PhD. Thesis, MIT Laboratory for Computer Science*, 1983.

[Ru '66] Rudin, Walter, **Real and Complex Analysis**, *McGraw-Hill*, 1996.

[SRL '90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky, **Priority Inheritance Protocols: An Approach to Real-Time Synchronization**, *IEEE Transactions on Computers* 39(9): 1175-1185, 1990.

[SSN '95] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo, **Implications of Classical Scheduling Results for Real-Time Systems**, *Computer*, 16-25, 1995.