A New Look at Pfair Priorities *

James H. Anderson and Anand Srinivasan

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
E-mail: {anderson,anands}@cs.unc.edu

September 1999

Abstract

We consider Pfair scheduling in real-time multiprocessor systems. Under Pfair scheduling, tasks are required to execute at steady rates. The most efficient Pfair scheduling algorithm proposed to date is an algorithm called PD developed by Baruah and colleagues. PD schedules periodic tasks by breaking them into quantum-length subtasks that are subject to intermediate deadlines. Ties among subtasks with the same intermediate deadline are broken by inspecting four tie-break parameters. PD improved upon a previous algorithm called PF, which relies on a less-efficient procedure for resolving ties.

In this paper, we show that the priority definition used in PD can be simplified to consist of one intermediate deadline and only two tie-break parameters. We also show that further simplifications are, in general, unlikely. In particular, we show if either tie-break parameter is eliminated, then there exists a feasible task set that is not correctly scheduled. Although both tie-breaks are needed in general, for the important special case of a two-processor system, we show that *no* tie-breaking information is required. In proving that our simplified version of PD is correct, we use an inductive "swapping" argument in which a schedule produced by an optimal scheduler is converted into one allowed by our priority definition by systematically interchanging pairs of subtasks. This proof reveals many fundamental properties inherent to Pfair scheduling.

Keywords: fairness, multiprocessors, optimality, Pfair, priorities, real-time scheduling.

*Work supported by NSF grants CCR 9732916 and CCR 9972211. The first author was also supported by an Alfred P. Sloan Research Fellowship.

1 Introduction

A major step forward in the evolution of processor scheduling techniques was recently achieved in the work of Baruah and colleagues on *Pfair scheduling* [4, 5, 6]. Pfair scheduling differs from more conventional real-time scheduling disciplines in that tasks are explicitly required to make progress at steady rates. In most real-time scheduling schemes, the notion of a rate is implicit. For example, in the classic periodic task model, a *period* $T.p$ and an *execution cost* $T.e$ are associated with each task T. Every $T.p$ time units, a new invocation of T with cost $T.e$ is released into the system, and each invocation of a task must complete execution before the next invocation of that task begins. In this model, each task T executes at a rate given by $T.e/T.p$. However, this notion of a rate is a bit inexact: during each interval of length $T.p$, there are no guarantees as to exactly *which* $T.e$ time units will be allocated to task T. In particular, an invocation of T may be allocated $T.e$ time units at the beginning of its period, or at the end of its period, or its computation may be spread out more evenly. Under Pfair scheduling, this implicit notion of a rate is strengthened to require each task to be executed at a rate that is uniform across each invocation.

Executing tasks at steady rates has important consequences. For instance, the Pfair scheduling algorithms proposed by Baruah et al. *optimally* solve the problem of scheduling periodic tasks on a multi-processor system in polynomial time. This is a problem that was previously viewed by most researchers as being almost undoubtedly *NP*-hard. Pfair scheduling algorithms schedule tasks by breaking them into quantum-length "subtasks" and by requiring each subtask to complete execution by an intermediate deadline. By breaking tasks into smaller executable units, Pfair scheduling algorithms circumvent many of the bin-packing-like problems that lie at the heart of intractability results involving multiple-resource real-time scheduling problems. Intuitively, it is easier to evenly distribute small, uniform items among the available bins than larger, nonuniform items.

Baruah et al. presented two Pfair scheduling algorithms called PF and PD [4, 5]. The two algorithms differ in the way in which ties are broken when two subtasks have the same intermediate deadline. In PF, ties are broken by comparing future intermediate deadlines, which is somewhat expensive. In PD, ties are broken in constant time by inspecting four tie-break parameters.

In this paper, we take a new look at the question of how to define subtask priorities in a Pfair-scheduled system. There are four main contributions of this paper. First, we show that the priority definition used in PD can be simplified to consist of one intermediate deadline and only two tie-break parameters. Both tie-break parameters are needed for reasons that are quite easily explained. Second, we present a collection of counterexamples that show that further simplifications are, in general, unlikely. In particular, we show if either of the two tie-break parameters is eliminated, then there exists a feasible task set that is not correctly scheduled. Third, although further simplifications are unlikely in general, for the important special case of a two-processor system, we show that *no* tie-breaking information is required.

The final main contribution of this paper is the proof of correctness we give for our simplified version of PD. In [5], PD is proved correct by means of a simulation argument that shows that PD "closely" tracks the behavior of PF. In contrast, we prove that our algorithm is correct by means of an inductive "swapping" argument in which a schedule produced by an optimal scheduler is converted into one allowed by our priority definition by systematically interchanging pairs of subtasks. This proof gives researchers interested in Pfair-scheduled systems a new set of techniques that can be applied to reason about such systems. We ourselves have been able to extend the arguments given in this paper to show that our algorithm can be applied to optimally schedule sporadic tasks [2]. In proving this result, much of the swapping proof presented in this paper was used intact. It is not obvious to us how one would modify the proof given in [5] for PD to deal with sporadic tasks. (Baruah has told us it is not obvious to him either.)

The rest of this paper is organized as follows. In Section 2, we present a brief overview of Pfair scheduling. Then, in Section 3, we present the priority definition that defines our Pfair scheduling algorithm. We prove that this algorithm is correct in Section 4. In Section 5, we show that our priority definition can be simplified on two-processor systems by eliminating all tie-breaking information. In Section 6, we present the counterexamples mentioned above, which show that our results are tight. We end with concluding remarks in Section 7.

2 Pfair Scheduling

Consider a collection of periodic real-time tasks to be executed on a system of multiple processors. We assume that processor time in such a system is allocated in discrete time units, or quanta; the time interval $[t, t+1)$, where t is a nonnegative integer, is called *slot* t. Associated with each task T is a *period* $T.p$ and an *execution cost* $T.e$. Every $T.p$ time units, a new invocation of T with a cost of $T.e$ time units is released into the system; we call such an invocation a *job* of T. Each job of a task must complete execution before the next job of that task begins. Thus, $T.e$ time units must be allocated to T in each interval $[k \cdot T.p, (k+1) \cdot T.p)$, where $k \geq 0$. T may be allocated time on different processors in such an interval, as long as it is not allocated time on different processors at the same time.

Following Baruah et al. [4], we refer to $T.e/T.p$ as the *weight* of task T. A task's weight defines the rate at which it is to be scheduled. Because processor time is allocated in quanta, we cannot guarantee that a task T will execute for *exactly* $(T.e/T.p)t$ time during each interval of length t. Instead, in a Pfair-scheduled system, processor time is allocated to each task T in a manner that ensures that its rate of execution never deviates too much from that given by its weight $T.e/T.p$. More precisely, correctness is defined by focusing on the *lag* between the amount of time allocated to each task and the amount of time that would be allocated to that task in an ideal system with a quantum approaching zero. Formally, the *lag of task* T *at time* t, denoted $lag(T, t)$, is defined as follows:

$$lag(T, t) = (T.e/T.p)t - allocated(T, t), \tag{1}$$

where $allocated(T, t)$ is the amount of processor time allocated to T in $[0, t)$. A schedule is *Pfair* if and only if

$$(\forall i, t :: -1 < lag(T, t) < 1). \tag{2}$$

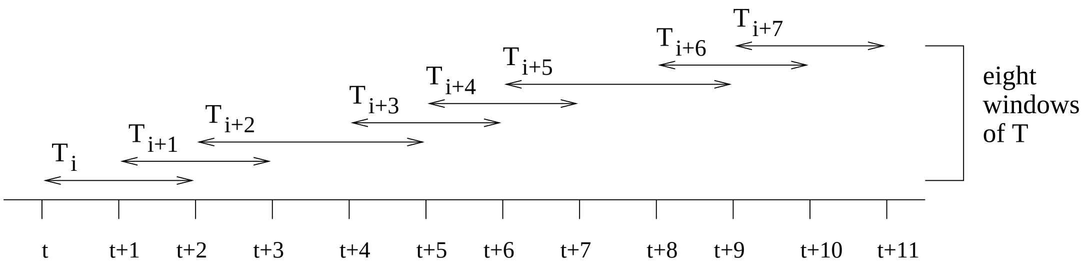


Figure 1: The eight "windows" of a task T with weight $T.e/T.p = 8/11$. Each of T's eight units of computation must be allocated processor time during its window, or else a lag-bound violation will result.

Informally, the allocation error associated with each task must always be less than one quantum. It is straightforward to show that any Pfair schedule is periodic. In particular, in a Pfair schedule, $lag(T, t) = 0$ for $t = 0, T.p, 2T.p, 3T.p, \ldots$. This is because, for these values of t, $(T.e/T.p)t$ is an integer, and therefore by (1), $lag(T, t)$ is an integer as well. By (2), if $lag(T, t)$ is an integer, then it must be 0.

The lag bounds given in (2) have the effect of breaking a task into smaller executable units that are subject to intermediate deadlines. To see this, consider a task T with weight $T.e/T.p = 8/11$. This task consists of eight units of computation, which we refer to as "subtasks". Suppose that a job of T is released at time t and let the subtasks of this instance be denoted $T_i, \ldots, T_{i+7}$, as shown in Figure 1. It can be shown that T_i *must* execute within either slot t or slot $t + 1$, or else T's lag bounds as given by (2) will be violated. In effect, there is a two-slot "window" during which T_i must be scheduled. In total, a task T with $T.e/T.p = 8/11$ will consist of eight windows as shown in Figure 1. Note that successive windows of T overlap by one slot. In general, consecutive windows of a task are either disjoint or overlap by one slot. (In Appendix C, a C program appears that can be used to generate the windows corresponding to tasks of various weights. Readers unfamiliar with Pfair scheduling may find it instructive to experiment with this program before continuing.)

Baruah et al. showed that a periodic task set τ has a Pfair schedule on M processors if and only if

$$\sum_{T \in \tau} \frac{T.e}{T.p} \leq M. \tag{3}$$

In addition, they presented two algorithms for producing Pfair schedules, called PF and PD, respectively [4, 5]. In PF, tasks are prioritized by focusing on the intermediate deadlines defined by their windows. We call these intermediate deadlines *pseudo-deadlines*. For example, subtask T_i in Figure 1 has a pseudo-deadline at time slot $t+1$ (the last slot into which it can be scheduled). If subtasks T_i and U_j are both "ready" to execute, then T_i is given priority over U_j if its pseudo-deadline is earlier than that of U_j. Some tie-breaking strategy must be used if several subtasks have equal pseudo-deadlines. PF breaks such a tie by considering subsequent pseudo-deadlines of each task. To give the reader some insight as to why this works, we briefly sketch the correctness proof for PF. Our intent here is to be as intuitive as possible; remaining formal definitions concerning Pfair-scheduled systems are given in the next section. For simplicity, we assume in this proof sketch that $T.p$ and $T.e$ are relatively prime for each task T. As explained in the next section, this is a reasonable assumption because, under Pfair scheduling, two tasks T and T' for which $T.e/T.p = T'.e/T'.p$ are scheduled in exactly the same way. If $T.e$ and $T.p$ are relatively prime, then it can be shown that the windows of any pair of consecutive subtasks of T overlap (by one slot) if and only if they are part of the same job. As we shall see next, the PF priority definition makes a distinction between consecutive windows that overlap and those that do not. By our assumption, consecutive windows that do not overlap occur only at job boundaries.

PF priorities: At time t, if subtasks T_i and U_j are both ready to execute, then T_i has higher priority than U_j if one of the following holds:

- T_i's pseudo-deadline is less than U_j's.
- T_i and U_j have equal pseudo-deadlines, neither is the final subtask of its job, and T_{i+1} has higher priority over U_{j+1}. (In essence, future subtasks of T and U are inductively considered in turn until the tie between T_i and U_j can be broken.)
- T_i and U_j have equal pseudo-deadlines and U_j is the final subtask of a job of U. (If T_i also is the final subtask of a job of T, then the tie between T_i and U_j can be broken arbitrarily.) □

The correctness proof for PF proceeds by inducting over the interval $(0, L]$, where L is the least common multiple of $\{T.p \mid T \in \tau\}$. The crux of the argument is to show the following: if there exists a Pfair schedule S such that all the scheduling decisions in S before slot t are in accordance with PF priorities, then there exists a Pfair schedule S' such that all the scheduling decisions in S' before slot $t+1$ are in accordance with PF priorities. If all scheduling decisions in S at slot t are in accordance with PF, then we can take S' to be S. Otherwise, we construct S' from S by means of a swapping argument in which some subtasks in S are interchanged. Suppose that, in S, **(i)** U_j is scheduled at slot t, **(ii)** T_i is ready to execute at t but is scheduled at a later slot, and **(iii)** T_i has higher priority than U_j at t. By the priority definition of PF, one of the following holds.

- **T_i's pseudo-deadline is less than U_j's.** Because T_i's pseudo-deadline is less than U_j's pseudo-deadline, and because the windows of consecutive subtasks overlap by at most one slot, U_{j+1} is scheduled at a later slot than T_i. Therefore, T_i and U_j can be directly swapped, as shown in Figure 2(a).
- **T_i and U_j have equal pseudo-deadlines, neither is the final subtask of its job, and T_{i+1} has higher priority than U_{j+1}.** The swapping shown in Figure 2(a) may not work here because T_i and U_{j+1} can be scheduled in the same slot. This can happen only if T_i is scheduled in the last slot of its window, as shown in Figure 2(b). To create a schedule in which T_i is scheduled in slot t, we first inductively swap T_{i+1} and U_{j+1}, as shown in Figure 2(b). (Note that this may cause future subtasks of T and U to be swapped as well.) This results in a schedule in which T_i and U_j can be safely swapped.
- **T_i and U_j have equal pseudo-deadlines and U_j is the final subtask of a job of U.** The swapping shown in Figure 2(a) is valid in this case too (although, in this case, U_j's pseudo-deadline is at slot t'

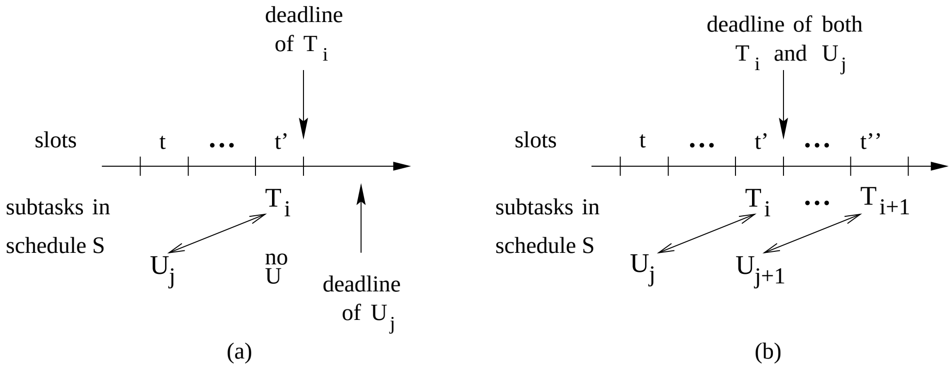


Figure 2: Correctness proof for PF. A double arrow indicates two subtasks that are to be swapped. In (a), "no U" means no subtask of U is scheduled in slot t'.

instead of later). In particular, because U_j is the final subtask of its job, the windows of U_j and U_{j+1} do not overlap. Therefore, U_{j+1} cannot be scheduled in the same slot as T_i.

The problem with PF is that it is inefficient: to determine which of two subtasks has higher priority, a vector of pseudo-deadlines must be compared. In PD, ties are resolved more efficiently by inspecting four tie-break parameters. We do not describe the four tie-break parameters used in PD here, as they are similar to those used in our algorithm. After presenting our algorithm, we discuss how it differs from PD.

Baruah et al. proved PD correct not by a swapping argument, but by a simulation argument that shows that PD "closely" tracks the behavior of PF. In contrast, we *do* use a swapping argument to show that our simplified version of PD is correct. Our proof is more complicated than the proof of PF because it involves multiple tasks. The key to our proof is a collection of properties involving subtasks and windows. These properties, which are proved in Appendix A, are used to show the existence of a valid swapping in all cases.

3 Simplified Pfair Scheduling Algorithm

In this section, we specify the priority definition used in our simplified Pfair scheduling algorithm. We begin by defining some additional terms that will be used in this and subsequent sections.

A *schedule* S is a mapping $S : \tau \times N \mapsto \{0, 1\}$, where τ is a set of periodic tasks and N is the set of natural numbers. If $S(T, t) = 1$, then we say that T *is scheduled at slot* t. S_t denotes the set of tasks scheduled in slot t. The statements $T \in S_t$ and $S(T, t) = 1$ are equivalent.

As stated earlier, each release of task T is called a *job* of T. We assume that $T.p$ (T's period) and $T.e$ (T's execution cost) satisfy $\frac{T.e}{T.p} < 1$ for each task T, because a task with $\frac{T.e}{T.p} = 1$ can be easily scheduled by assigning it to a dedicated processor. A task with weight less than $1/2$ is called a *light* task, while a task with weight at least $1/2$ is called a *heavy* task.

Windows and related definitions. The lag bounds given in (2) have the effect of breaking each task T into an infinite sequence of unit-time *subtasks*. We denote the i^{th} subtask of task T as T_i, where $i \geq 1$. As in [4], we associate with each subtask T_i a *pseudo-release*

$$r(T_i) = \left\lfloor \frac{(i-1) \cdot T.p}{T.e} \right\rfloor \tag{4}$$

and a *pseudo-deadline*

$$d(T_i) = \left\lceil \frac{i \cdot T.p}{T.e} \right\rceil - 1. \tag{5}$$

(These expressions are actually derived in Appendix A.) The interval $[r(T_i), d(T_i)]$ is called the *window* of subtask T_i and is denoted by $w(T_i)$. $r(T_i)$ is the first slot into which T_i could potentially be scheduled, and $d(T_i)$ is the last such slot. For example, in Figure 1, $r(T_i) = t$ and $d(T_i) = t + 1$. For brevity, we often refer to pseudo-deadlines and pseudo-releases as simply deadlines and releases, respectively. We define subtask T_i to be *ready* at time $t \in w(T_i)$, denoted $subtask(T,t) = i$, if T_{i-1} has been scheduled prior to t but T_i has not. We sometimes write $T_i \in S_t$ as an abbreviation for $T \in S_t \ \wedge \ subtask(T,t) = i$. The *length* of window $w(T_i)$, denoted by $|w(T_i)|$, is defined as $d(T_i) - r(T_i) + 1$. A window spanning n time slots is called an *n-window*.

Claim 1 *If T and T' are two tasks such that $\frac{T.e}{T.p} = \frac{T'.e}{T'.p}$, then $r(T_i) = r(T'_i)$ and $d(T_i) = d(T'_i)$ for all $i \geq 1$.*

Proof: Follows directly from (4) and (5). □

Given Claim 1, we henceforth make the simplifying assumption that $T.e$ and $T.p$ are relatively prime for each task T.

(4) and (5) imply that $r(T_{i+1})$ is either $d(T_i)$ or $d(T_i)+1$, i.e., successive windows are either disjoint or overlap by one slot. We define a bit $b(T_i)$ that distinguishes between these two possibilities.

$$b(T_i) = \begin{cases} 1, & \text{if } r(T_{i+1}) = d(T_i) \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Claim 2 *$b(T_i) = 0$ if and only if $w(T_i)$ is the last window of a job of T.*

Proof: "If": If $w(T_i)$ is the last window of a job of T, then i is a multiple of $T.e$. Therefore, $r(T_{i+1}) = d(T_i) + 1 = k \cdot T.p$, where $k = \frac{i}{T.e}$. It follows that $b(T_i) = 0$.

"Only If": If $b(T_i) = 0$, then by the definition of b, $r(T_{i+1}) = d(T_i) + 1$. Therefore, by (4) and (5), $\left\lfloor \frac{i \cdot T.p}{T.e} \right\rfloor = \left\lceil \frac{i \cdot T.p}{T.e} \right\rceil$. This implies that $\frac{i \cdot T.p}{T.e}$ is an integer. Because we have assumed that $T.p$ and $T.e$ are relatively prime, it follows that i is a multiple of $T.e$. Thus, $w(T_i)$ is the last window of a job of T. □

The following properties concerning windows are proved in Appendix A.

(P1) The windows of each task T are symmetric within each job of T, i.e., $|w(T_{ke+i})| = |w(T_{ke+e+1-i})|$, where $e = T.e$, $1 \leq i \leq T.e$, and $k \geq 0$.

(P2) The length of each of task T's windows is either $\left\lceil \frac{T.p}{T.e} \right\rceil$ or $\left\lceil \frac{T.p}{T.e} \right\rceil + 1$. A window of task T with length $\left\lceil \frac{T.p}{T.e} \right\rceil$ (respectively, $\left\lceil \frac{T.p}{T.e} \right\rceil$+1) is called a *minimal* (respectively, *maximal*) window of T.

(P3) The first window of each job of a task is a minimal window of that task.

(P4) A task has a 2-window if and only if it is heavy. By (P1) and (P3), the first and last windows of any job of a heavy task are both of length two.

Group deadlines. Consider a sequence $T_i, \ldots, T_j$ of subtasks of a heavy task T such that $|w(T_k)| = 2 \ \wedge\ b(T_k) = 1$ for all $i < k \leq j$ and either $|w(T_{j+1})| = 3$ or $w(T_{j+1}) = 2 \ \wedge\ b(T_{j+1}) = 0$ (e.g., T_i, T_{i+1} or T_{i+2}, T_{i+3}, T_{i+4} or T_{i+5}, T_{i+6} in Figure 1). If any one of the subtasks $T_i, \ldots, T_j$ is scheduled in the last slot of its window, then each subsequent subtask in this sequence must be scheduled in its last slot. In effect, $T_i, \ldots, T_j$ must be considered as a single schedulable entity subject to a "group" deadline. Formally, we define $d(T_j) + 1$ to be the *group deadline* for the group of subtasks $T_i, \ldots, T_j$. Intuitively, if we imagine a job of T in which each subtask is scheduled in the first slot of its window, then the slots that remain empty exactly correspond to the group deadlines of T. For example, in Figure 1, T has group deadlines at slots $t + 3$, $t + 7$, and $t + 10$. Note that each group deadline of a heavy task is either the middle slot of a 3-window or the second slot of the final 2-window of some job. The following properties concerning group deadlines are proved in Appendix A.

(P5) If t and t' are successive group deadlines of a task T, then $t' - t$ is either $\left\lfloor \frac{T.p}{T.p - T.e} \right\rfloor$ or $\left\lfloor \frac{T.p}{T.p - T.e} \right\rfloor + 1$.

(P6) Let T be a heavy task with more than one group deadline per job. Let t and t' (respectively, u and u') be successive group deadlines of T, where t' (respectively, u') is the first (respectively, last) group deadline within a job of T (for the first job of T, take t to be -1). Then, $t' - t = u' - u + 1$.

We let $D(T_i)$ denote the group deadline of subtask T_i. Formally, if T is heavy, then

$$D(T_i) = (\mathbf{min}\ u :: u > d(T_i) \text{ and } u \text{ is a group deadline of } T).$$

For example, in Figure 1, $D(T_i) = t + 3$ and $D(T_{i+5}) = t + 10$. (Note that, according to this definition, T_{i+7} in Figure 1 would be assigned a group deadline within the next job of T. It actually turns out that group deadlines are not needed to break a tie involving a subtask like T_{i+7}, which occurs at the end of its job.) The above definition of D is valid only for heavy tasks. If T is light, then $D(T_i) = 0$.

Priority Definition: Task T's priority at time t is defined to be $(d(T_i), b(T_i), D(T_i))$, where $subtask(T, t) = i$. Priorities are ordered according to the following relation.

$$(d', b', D') \ \preceq\ (d, b, D) \ \equiv\ [d < d'] \ \vee\ [(d = d') \ \wedge\ (b > b')] \ \vee\ [(d = d') \ \wedge\ (b = b') \ \wedge\ (D \geq D')]$$

If $subtask(T, t) = i$ and $subtask(U, t) = j$, then T's priority is at least U's at time t if $(d(U_j), b(U_j), D(U_j)) \preceq (d(T_i), b(T_i), D(T_i))$. □

According to this definition, T has higher priority than U if the pseudo-deadline of T's current subtask is less than that of U's. If two subtasks have the same pseudo-deadline, then the bit that we defined for each pseudo-deadline is used as a tie-breaker. The reason for this is the same as with PF, i.e., to distinguish between pseudo-deadlines that occur within a job and those that occur at the end of a job. If two heavy subtasks have equal pseudo-deadlines and the bits associated with them are the same, then the subtask with the greater group deadline has higher priority. Note that, due to the definition of D, if a heavy subtask and a light subtask have identical pseudo-deadlines and associated bits, then the tie is always resolved in favor of the heavy subtask. Also note that if a set of light-only tasks is to be scheduled, then the D parameter is not needed. Finally, note that it is possible for two tasks to have identical priorities; such ties can be broken arbitrarily.

Given our priority definition, the mechanics of producing a schedule on-line are exactly as with PD. Thus, the scheduling algorithm proposed in [5] can be used (this algorithm is described in Appendix B). A task's next pseudo-deadline and b-bit tie-break can be computed at run-time in constant time using Equations (4), (5), and (6). A task's next group deadline can also be computed in constant time, using Equation (32) in Appendix A.

The priority definition used in PD is similar to ours, except that two additional tie-break parameters are used. The first of these is a task's weight. The second is a bit associated with each group deadline that is similar to the b bit we associate with pseudo-deadlines. For example, in Figure 1, the bit corresponding to $D(T_i)$ would be defined to be 1 and the bit corresponding to $D(T_{i+5})$ would be defined to be 0. These bits are used to distinguish between group deadlines that correspond to a 3-window and ones that correspond to a final 2-window of a job. Our results show that these two additional tie-break parameters are not needed.

4 Correctness Proof

We now prove that our algorithm produces a Pfair schedule. Throughout this section, we assume that $\sum_{T\in\tau} \frac{T.e}{T.p} = M$ (refer to (3)), where M is the number of processors. If less than M, then several dummy tasks can be added to make $\sum_{T\in\tau} \frac{T.e}{T.p} = M$. Like the PF proof sketched earlier, correctness is established by showing that if there exists a Pfair schedule S for which scheduling decisions are in accordance with our priority definition up to time $t-1$, then there exists a Pfair schedule for which scheduling decisions are in accordance with our priority definition up to time t. The number of scheduling decisions in S at time t that are *not* in accordance with our priority definition can be inductively reduced to zero by interchanging (swapping) some subtasks. Such interchanges will be valid if all subtasks are still scheduled within their windows, no two subtasks of the same task are scheduled in the same time slot, and M tasks are scheduled in each time slot.

We now state and prove two lemmas. The second of these, Lemma 2, gives the inductive step of our correctness proof. Lemma 1 deals with a situation that arises in one of the cases in Lemma 2. According to Lemma 1, if subtasks T_i, U_j, and U_{j+1} are scheduled as shown in Figure 3(a), then a valid swapping exists that moves U_j out of slot t.

Lemma 1 *Let S be a Pfair schedule such that for light tasks T and U and $t < t'$, $subtask(T,t) = i$, $subtask(U,t)$*

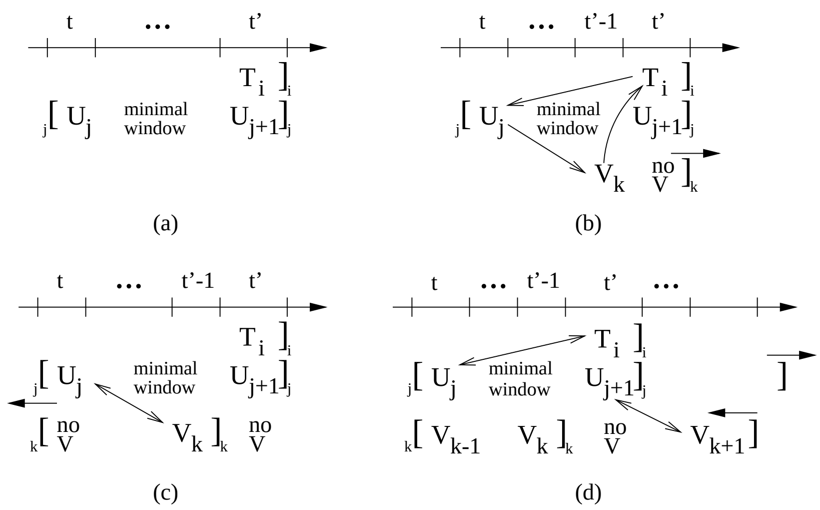


Figure 3: We use the following notation in this and subsequent figures. "[" and "]" indicate the release and deadline of a subtask; subscripts indicate which subtask. Each task is shown on a separate line. An arrow from subtask T_i to subtask U_j indicates that T_i is now scheduled in place of U_j. An arrow over "[" (or "]") indicates that the actual position of "[" (or "]") can be anywhere in the direction of the arrow. Time is divided into unit-time slots that are numbered. (Although all slots are actually of the same length, due to formatting concerns, they don't necessarily appear as such in our figures.) If T_i is released at slot t, then "[" is aligned with the left side of slot t. If T_i has a deadline at slot t, then "]" is aligned with the right side of slot t. **(a)** Conditions of Lemma 1. **(b)** $d(V_k) > t' - 1$. **(c)** $d(V_k) = t' - 1$ and $r(V_k) \leq t$. **(d)** $d(V_k) = t' - 1$ and $r(V_k) \geq t$.

$= j$, $U_j \in S_t$, $U_{j+1} \in S_{t'}$, $T_i \in S_{t'}$, $r(U_j) = t$, $d(U_j) = t'$, $r(U_{j+1}) = t'$, $d(T_i) = t'$, where $w(U_j)$ is a minimal window of U. Then, there exists a Pfair schedule S' such that $U \notin S'_t$, $S_u = S'_u$ for $0 \leq u < t$, and $S_t - \{U\} \subset S'_t$.

Proof: Note that T_i and U_j cannot be swapped directly because this would result in a schedule in which two subtasks of U are scheduled in the same slot. Instead, we identify another subtask V_k that can be used as an intermediate between U_j and T_i for swapping. Because T and U are both light, by (P2), all windows of each span at least three slots. Thus, $t' > t + 1$ and $\{T, U\} \not\subseteq S_{t'-1}$. Because all processors are fully utilized and $\{T, U\} \not\subseteq S_{t'-1}$ and $\{T, U\} \subseteq S_{t'}$, there exists a task V such that $V \in S_{t'-1}$ and $V \notin S_{t'}$. Let $k = subtask(V, t' - 1)$. If $d(V_k) > t' - 1$, then the swapping shown in Figure 3(b) gives the desired schedule. In the rest of the proof, we assume

$$d(V_k) = t' - 1. \tag{7}$$

If $r(V_k) < t$ or if $r(V_k) = t \wedge V \notin S_t$, then the swapping shown in Figure 3(c) produces the desired schedule. The remaining possibility to consider is

$$(r(V_k) > t) \vee (r(V_k) = t \wedge V \in S_t). \tag{8}$$

In this case, we show that the swapping in Figure 3(d) is valid (this figure actually depicts the case $r(V_k) = t \wedge V \in S_t$). From (7), (8), and the statement of the lemma, we have $d(V_k) = d(U_j) - 1$ and $r(V_k) \geq r(U_j)$.

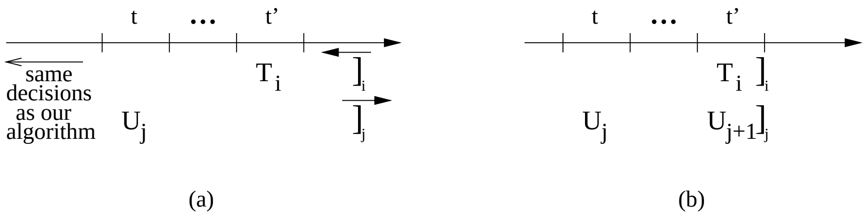


Figure 4: **(a)** Conditions of Lemma 2. **(b)** The "difficult" case to consider.

This implies that

$$|w(V_k)| < |w(U_j)|. \tag{9}$$

Because $w(U_j)$ is a minimal window of U, $|w(U_{j+1})| \geq |w(U_j)|$. By definition, $|w(U_{j+1})| = d(U_{j+1}) - r(U_{j+1}) + 1$. From the statement of the lemma, this implies that $d(U_{j+1}) = |w(U_{j+1})| + t' - 1$. Therefore,

$$d(U_{j+1}) \geq t' - 1 + |w(U_j)|. \tag{10}$$

Now, by (7) and the fact that consecutive windows of a task overlap by at most one slot, V_{k+1} is released at either $t' - 1$ or t'. We now show that in either case, $d(V_{k+1}) \leq t' - 1 + |w(V_k)|$. If $r(V_{k+1}) = t'$, then by (P1), $|w(V_{k+1})| = |w(V_k)|$. By definition, $|w(V_{k+1})| = d(V_{k+1}) - r(V_{k+1}) + 1$. Therefore, $d(V_{k+1}) = t' - 1 + |w(V_k)|$.

On the other hand, if $r(V_{k+1}) = t' - 1$, then we reason as follows. By (P2), $|w(V_{k+1})| \leq |w(V_k)| + 1$. Because $|w(V_{k+1})| = d(V_{k+1}) - r(V_{k+1}) + 1$, it follows that $d(V_{k+1}) \leq t' - 1 + |w(V_k)|$. Hence, in both cases, $d(V_{k+1}) \leq t' - 1 + |w(V_k)|$. By (9) and (10), this implies that $d(U_{j+1}) > d(V_{k+1})$. Thus, the swapping shown in Figure 3(d) is valid, and produces the required schedule. □

We now state and prove Lemma 2; the conditions of the lemma are depicted in Figure 4(a).

Lemma 2 *Let S be a Pfair schedule such that all scheduling decisions before slot t are in accordance with our priority definition. Let A denote the set of tasks that would be scheduled in slot t according to our priority definition. Suppose $A \neq S_t$. Let T be a task of highest priority in $A - S_t$, and let U be a task of lowest priority in $S_t - A$. Then, there exists a Pfair schedule S' such that $S'_u = S_u$ for $0 \leq u < t$ and $S'_t = (S_t - \{U\} + \{T\})$.*

Proof: Let $i = subtask(T, t)$ and $j = subtask(U, t)$. Because S is a valid schedule, there exists t' such that $t < t' \leq d(T_i) \ \wedge \ T_i \in S_{t'} \ \wedge \ (\forall u : t \leq u < t' :: T \notin S_u)$. If $d(T_i) < d(U_j)$ or if $d(T_i) = d(U_j) \ \wedge \ b(U_j) = 0$, then no subtask of U can be scheduled in the interval $(t, t']$. Therefore, T_i and U_j can be directly swapped to get the required schedule. In the remainder of the proof, we assume that

$$d(T_i) = d(U_j) \ \wedge \ b(U_j) = 1. \tag{11}$$

Because T has higher priority than U at time t, we have

$$b(T_i) = 1. \tag{12}$$

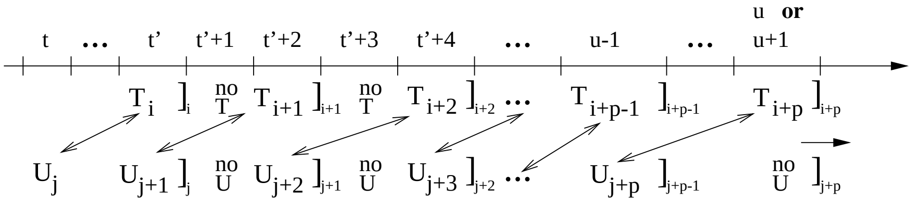


Figure 5: Case 2. T is heavy, U is light, and $d(T_i) = d(U_j)$.

If $U_{j+1} \notin S_{t'}$, then T and U still can be directly swapped to get a valid schedule. Thus, we henceforth assume $U_{j+1} \in S_{t'}$, i.e., T_i and U_{j+1} are both scheduled in slot t'. By (11), this can happen only if $d(T_i)$ and $d(U_j)$ both equal t' (otherwise, $w(U_j)$ and $w(U_{j+1})$ would overlap by more than one slot). Thus, we have the following.

$$U_{j+1} \in S_{t'} \ \wedge \ d(T_i) = t' \ \wedge \ d(U_j) = t'. \tag{13}$$

We now consider four cases, which depend on the weights of tasks T and U. We remind the reader that, in all of these cases, (11) through (13) are assumed to hold. These conditions are depicted in Figure 4(b).

Case 1: *T* is light and *U* is heavy. By the definition of D, T cannot have higher priority than U at time t.

Case 2: *T* is heavy and *U* is light. In this case, we show that the swapping in Figure 5 is valid. By (P2) and (P4), all windows of T are of length two or three, and the last window of each job of T is of length two. By (12), $w(T_i)$ is not the final window of its job. Thus, there exists r such that

$$|w(T_{i+r})| = 2 \ \wedge \ (\forall k : 0 < k < r :: |w(T_{i+k})| = 3 \ \wedge \ b(T_{i+k}) = 1).$$

(Note that r could be one, i.e., T could have no three-windows between $w(T_i)$ and $w(T_{i+1})$.) Because U is light, by (P4), $|w(U_k)| \geq 3$ for all k. This implies that $d(T_{i+r}) < d(U_{j+r})$. Let q denote the smallest value of k that satisfies $d(T_{i+k}) < d(U_{j+k})$. (Note that $q \leq r$.) Then,

$$(\forall k : 0 < k < q :: d(T_{i+k}) = d(U_{j+k}) \ \wedge \ |w(T_{i+k})| = 3 \ \wedge \ |w(U_{j+k})| = 3 \ \wedge \ b(T_{i+k}) = 1) \ \wedge \ d(T_{i+q}) < d(U_{j+q}).$$

Because $d(T_{i+q}) < d(U_{j+q})$, we have $d(T_{i+q}) < r(U_{j+q+1})$. Thus, T_{i+q} is scheduled before U_{j+q+1}. Let p be the smallest value for k such that T_{i+k} is scheduled prior to U_{j+k+1}. (Again, note that $p \leq q$). To summarize, we have

- $(\forall k : 0 < k < p :: d(T_{i+k}) = d(U_{j+k}) \wedge |w(T_{i+k})| = 3 \wedge |w(U_{j+k})| = 3 \wedge b(T_{i+k}) = 1) \wedge d(T_{i+p}) \leq d(U_{j+p})$,
- T_{i+p} is scheduled before U_{j+p+1}, and

- for each k in the range $0 < k < p$, T_{i+k} is *not* scheduled before U_{j+k+1}.

It is straightforward to see that the relevant subtasks must be scheduled as shown in Figure 5. Thus, the swapping shown in this figure is valid.

Case 3: Both T and U are light. (This case and Case 4 are somewhat lengthy.) We show that one of the swappings in Figure 6 is valid. Because U is light, by (P4), $|w(U_{j+1})| \geq 3$. Therefore, U_{j+2} is released after $t' + 1$ (refer to Figure 4(b)), and so $U \notin S_{t'+1}$. Therefore, because $U \in S_{t'}$, and because all processors are fully utilized, there exists a task V such that $V \notin S_{t'}$ and $V \in S_{t'+1}$. Let $k = \mathit{subtask}(V, t'+1)$. Because V_k is scheduled at time $t'+1$, we have $r(V_k) \leq t'+1$. If $r(V_k) < t'+1$, then the swapping shown in Figure 6(a) produces the desired schedule. In the rest of the proof for Case 3, we assume

$$r(V_k) = t' + 1, \tag{14}$$

in which case this swapping is not valid. If V_{k-1} is scheduled in the interval (t, t'), then the swapping shown in 6(b) is valid. If V_{k-1} is not scheduled in (t, t'), then it is scheduled at or before t. We now show that it cannot be scheduled in slot t.

Suppose, to the contrary, that V_{k-1} *is* scheduled in slot t, as depicted in Figure 6(c). Because $r(V_k) = t'+1$, $d(V_{k-1})$ is either $t'+1$ or t'. If $d(V_{k-1})$ is t', then $b(V_{k-1}) = 0$. In either case, V_{k-1} has lower priority than U_j at t. Hence, because U_j is in $S_t - A$ (i.e., it is not among the subtasks that would be scheduled at time t according to our priority definition), V_{k-1} must be in $S_t - A$ as well. However, this contradicts our choice of U_j as a subtask of lowest priority in $S_t - A$. Thus, V_{k-1} is not scheduled in slot t.

In the rest of the proof for Case 3, we consider the remaining possibility, i.e., that V_{k-1} is scheduled at a time $v < t$. Now, it must be the case that T_i was not eligible to be scheduled at time v. To see this, note that if T_i *were* eligible at time v, then according to our priority definition, it should have been scheduled at time v, as T_i has higher priority than V_{k-1} (the reasoning here is similar to that in the previous paragraph). Thus, either $r(T_i) > v$ or $r(T_i) = v \ \wedge\ T_{i-1} \in S_v$. Because $r(V_{k-1}) \leq v$, this implies that either $(r(T_i) > r(V_{k-1}))$ or $(r(T_i) = v \ \wedge\ r(V_{k-1}) = v \ \wedge\ T_{i-1} \in S_v)$. We consider these two subcases next.

Subcase 3.A: $r(T_i) > r(V_{k-1})$. We show that $d(V_k) > d(T_{i+1})$, which implies that the swapping in Figure 6(d) is valid. There are two possibilities to consider, depending on the value of $b(V_{k-1})$.

$\underline{b(V_{k-1}) = 0}$. In this case, by the definition of $b(V_{k-1})$, we have $d(V_{k-1}) = r(V_k) - 1$. By (13) and (14), this implies that $d(V_{k-1}) = d(T_i)$. Since $b(V_{k-1})$ is 0, $w(V_{k-1})$ is the last window of its job, and $w(V_k)$ is the first window of its job. Thus, by (P1), $|w(V_k)| = |w(V_{k-1})|$. Because $r(V_{k-1}) < r(T_i)$ (our assumption for Subcase 3.A) and $d(V_{k-1}) = d(T_i)$, we have $|w(T_i)| \leq |w(V_{k-1})| - 1$. Therefore, $|w(T_i)| \leq |w(V_k)| - 1$. By (P2), $|w(T_{i+1})| \leq |w(T_i)| + 1$, and hence, $|w(V_k)| \geq |w(T_{i+1})|$. Because $r(V_k) = r(T_{i+1}) + 1$ (see Figure 6(d)), this implies that $d(V_k) > d(T_{i+1})$.

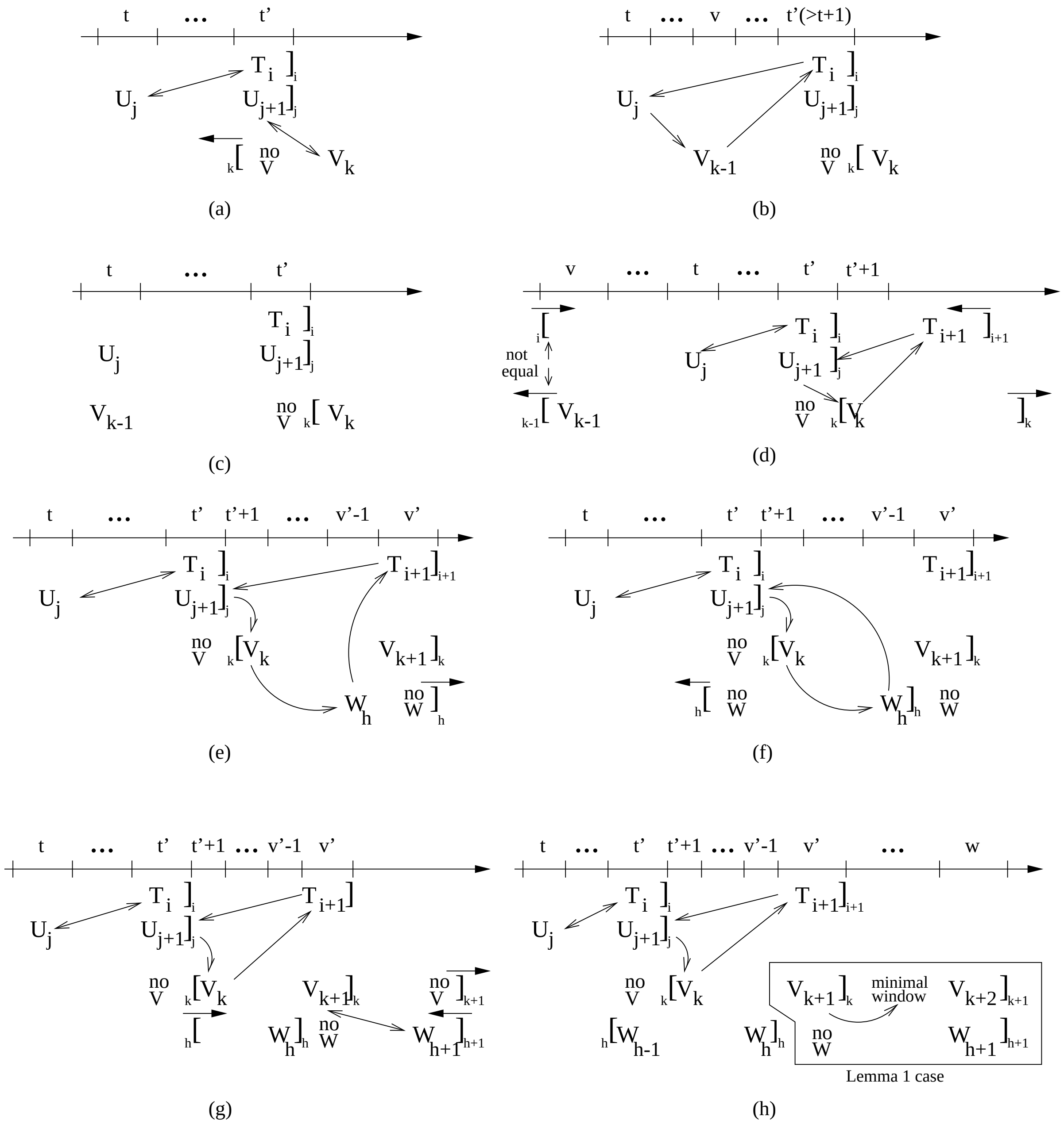


Figure 6: Case 3. **(a)** $r(V_k) \leq t+1$. **(b)** $r(V_k) = t+2 \wedge r(V_{k-1}) < r(T_i)$. **(c)** $d(W_h) \geq v'$. **(d)** $d(W_h) = v'-1 \wedge r(W_h) \leq t+1 \wedge W \notin S_{t+1}$. **(e)** $d(W_h) = v'-1 \wedge r(W_h) \geq t+2$. **(f)** $d(W_h) = v'-1$, $r(W_h) = t+1$, and $W \in S_{t+1}$.

<u>$b(V_{k-1}) = 1$.</u> In this case, by the definition of $b(V_{k-1})$, we have $d(V_{k-1}) = r(V_k)$. By (13) and (14), this implies that $d(V_{k-1}) = d(T_i)+1$. Because $r(V_{k-1}) < r(T_i)$ (our assumption for Subcase 3.A) and $d(V_{k-1}) = d(T_i)+1$, we have

$$|w(V_{k-1})| \geq |w(T_i)| + 2. \tag{15}$$

By (P2), $|w(V_k)| \geq |w(V_{k-1})|-1$ and $|w(T_i)| \geq |w(T_{i+1})|-1$. Hence, by (15), $|w(V_k)|+1 \geq |w(T_{i+1})|-1+2$, i.e., $|w(V_k)| \geq |w(T_{i+1})|$. Because $r(V_k) = r(T_{i+1}) + 1$ (again, see Figure 6(d)), this implies that $d(V_k) > d(T_{i+1})$.

Subcase 3.B: $r(T_i) = v \ \wedge \ r(V_{k-1}) = v \ \wedge \ T_{i-1} \in S_v$**.** Reasoning as in Subcase 3.A, it is possible to show that

$$d(V_k) \geq d(T_{i+1}). \tag{16}$$

We now show that a valid swapping exists in all cases. First, note that if T_{i+1} is scheduled before V_{k+1}, then the swapping shown in Figure 6(d) is still valid. This will be the case if $d(V_k) > d(T_{i+1})$ or if $d(V_k) = d(T_{i+1}) \ \wedge \ r(V_{k+1}) = d(V_k) + 1$. In the rest of the proof for Subcase 3.B, we assume that T_{i+1} is *not* scheduled before V_{k+1}. By (16), this can happen only if there exists v' such that

$$d(V_k) = v' \ \wedge \ r(V_{k+1}) = v' \ \wedge \ d(T_{i+1}) = v' \ \wedge \ V_{k+1} \in S_{v'} \ \wedge \ T_{i+1} \in S_{v'}. \tag{17}$$

Before considering other possible swappings, we first show that $w(V_k)$ is a minimal window of V; this fact is used several times in the reasoning that follows. By (14) and the fact that consecutive windows of the same task overlap by at most one slot, $d(V_{k-1})$ is either t' or $t'+1$. If $d(V_{k-1}) = t'$, in which case $w(V_{k-1})$ and $w(V_k)$ do not overlap, then $w(V_k)$ is the first window of its job. Hence, by (P3), $w(V_k)$ is a minimal window. On the other hand, if $d(V_{k-1}) = t'+1$, in which case $w(V_{k-1})$ and $w(V_k)$ do overlap, then we have the following:

- T_i and V_{k-1} are both released at slot v (our assumption for Subcase 3.B),
- T_i has a deadline at t' (see (13)) and V_{k-1} has a deadline at $t'+1$ (by assumption), and
- T_{i+1} and V_k have equal deadlines (by (17)).

By (P1)-(P3), this can happen only if $|w(V_k)| = |w(V_{k-1})| - 1$, which implies that $w(V_k)$ is a minimal window of V.

Now, because T_i and V_{k-1} are both released at slot v (our assumption for Subcase 3.B), and because T_i has a deadline at t' and V_k is released at $t'+1$ (see (13) and (14)), either $|w(V_{k-1})| = |w(T_i)| + 1$ or $|w(V_{k-1})| = |w(T_i)| \ \wedge \ b(V_{k-1}) = 0$. In either case, because T is light, by (P2)-(P4), V must also be light, and hence $|w(V_k)| \geq 3$. By (14) and (17), $w(V_k) = [t'+1, v']$; hence, $v' \geq t'+3$. Because $V_k \in S_{t'+1} \ \wedge \ V_{k+1} \in S_{v'}$, this implies that $V \notin S_{v'-1}$. Thus, by our assumption that all processors are fully utilized, there exists a task W such that $W \in S_{v'-1}$ and $W \notin S_{v'}$. Let $h = \mathit{subtask}(W, v'-1)$. We now show that at least one of the swappings in Figure 6(e)-(h) is valid.

$\underline{d(W_h) \geq v'}$**.** In this case, the swapping in Figure 6(e) is clearly valid. We henceforth assume

$$d(W_h) = v' - 1. \tag{18}$$

$\underline{(r(W_h) < t') \ \vee \ (r(W_h) = t' \ \wedge \ W \notin S_{t'})}$. In this case, the swapping shown in Figure 6(f) is valid.

$\underline{r(W_h) > t'}$**.** In this case, we show that $d(W_{h+1}) < d(V_{k+1})$, which implies that the swapping in Figure 6(g) is valid. $r(W_h) > t'$ implies that $|w(W_h)| < |w(V_k)|$ (see Figure 6(g)). Because $w(V_k)$ is a minimal window of V (as shown above), $|w(V_k)| \leq |w(V_{k+1})|$. Thus,

$$|w(W_h)| < |w(V_{k+1})|. \tag{19}$$

Now, consider $b(W_h)$. If $b(W_h) = 0$, then by (18), $r(W_{h+1}) = v'$, which by (17) implies that $r(W_{h+1}) = r(V_{k+1})$. In addition, by (P1), $|w(W_{h+1})| = |w(W_h)|$. Hence, by (19), we have $|w(W_{h+1})| < |w(V_{k+1})|$. Therefore, $d(W_{h+1}) < d(V_{k+1})$.

If $b(W_h) = 1$, then by (18), $r(W_{h+1}) = v' - 1$, which by (17) implies that $r(W_{h+1}) < r(V_{k+1})$. In addition, by (P2), $|w(W_{h+1})| \leq |w(W_h)|+1$. Hence, by (19), we have $|w(W_{h+1})| \leq |w(V_{k+1})|$. Therefore, $d(W_{h+1}) < d(V_{k+1})$.

$\underline{r(W_h) = t' \ \wedge \ W \in S_{t'}}$**.** In this case, analysis similar to that above shows that $d(W_{h+1}) \leq d(V_{k+1})$. Let $d(W_{h+1}) = w$. If $d(W_{h+1}) < d(V_{k+1})$ or if $d(W_{h+1}) = d(V_{k+1}) \ \wedge \ V_{k+2} \notin S_w$, then the swapping shown in Figure 6(g) is valid (the figure actually shows W_h being released after time t', but the swapping is still valid). On the other hand, if $d(W_{h+1}) = d(V_{k+1})$ and $V_{k+2} \in S_w$, then we have the following (see Figure 6(h)):

- W_h is released at slot t' and V_k is released at slot $t' + 1$,
- V_{k+1} is released at slot v', and
- V_{k+1} and W_{h+1} have deadlines at slot w.

By (P1)-(P3), this can happen only if $|w(V_{k+1})| = |w(V_k)|$. Because $w(V_k)$ is a minimal window of V, this implies that $w(V_{k+1})$ is a minimal window as well. Thus, by Lemma 1, there exists a schedule in which V_{k+1} is not scheduled at time v'. The swapping shown in Figure 6(h) is therefore valid. This exhausts all the possibilities if T and U are both light.

Case 4: Both T and U are heavy. In the proof for this case, we refer to successive group deadlines of a task. The following notation will be used. If g is a group deadline of task X, then $pred(X, g)$ (respectively, $succ(X, g)$) denotes the group deadline of task X that occurs immediately before (respectively, after) g. For example, in Figure 1, $pred(T, t+7) = t+3$ and $succ(T, t+7) = t+10$.

As before, we are dealing with the situation depicted in Figure 4(b). Because T_i has higher priority than U_j at time t according to our priority definition, $D(U_j) \leq D(T_i)$. Because $T_i \in S_{t'}$ and $t' = d(T_i)$ (refer to Figure 4(b)), each subsequent subtask of T with a deadline at or before $D(T_i)$ is scheduled in the last slot of its window. Note that, because T and U are heavy tasks, t' is either $t+1$ or $t+2$. Let u be the earliest time after t' such that $U \notin S_u$. Then, $u \leq D(U_j)$. Because $D(U_j) \leq D(T_i)$, this implies that either $u < D(T_i)$ or $u = D(T_i)$. If $u < D(T_i)$ holds then we have the following (refer to Figure 7(a)):

- in all slots in $[t', u]$, a subtask of T is scheduled in the last slot of its window;

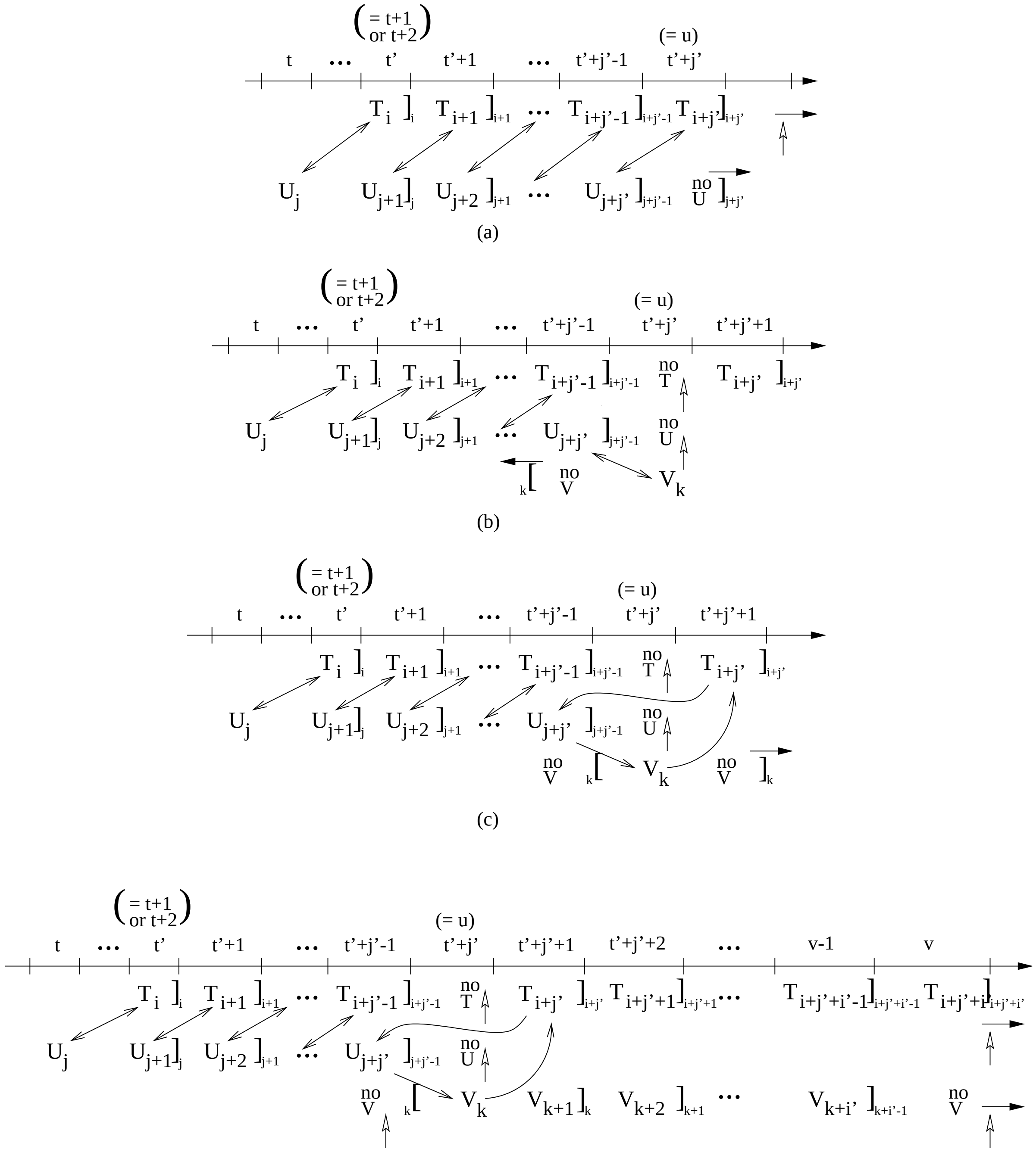


Figure 7: Case 4. We use the following notation in Figures 7-9. A group deadline at slot t is denoted by an up-arrow that is aligned with the right side of slot t. A left- or right-pointing arrow over an up-arrow indicates a group deadline that may be anywhere in the direction of the left- or right-pointing arrow. **(a)** $D(T_i) > D(U_j)$ or $D(T_i) = D(U_j) \ \wedge \ T \in S_u$. **(b)** $D(T_i) = D(U_j)$ and $r(V_k) < t' + j'$. **(c)** $D(T_i) = D(U_j)$, $r(V_k) = t' + j'$, and $V \notin S'_{t'+j'+1}$. **(d)** $D(T_i) = D(U_j)$, $r(V_k) = t' + j'$, and $D(T_{i+j'}) > v$.

- in all slots in $[t', u-1]$, a subtask of U is scheduled in the first slot of its window;
- no subtask of U is scheduled in slot u.

This implies that the swapping in Figure 7(a) is valid. If $u = D(T_i) \wedge T \in S_u$, then a similar swapping is valid (in this case, the subtasks of T to be swapped occupy all slots in the interval $[t', D(T_i)]$).

The remaining possibility to consider is $u = D(T_i) \wedge T \notin S_u$. In this case, because $u \leq D(U_j) \leq D(T_i)$, we have the following.

$$D(T_i) = D(U_j) \wedge D(U_j) = u \wedge T \notin S_u$$

Let $j + j' = \mathit{subtask}(U, u-1)$. Then, $u = t' + j'$, as shown in Figure 7(b)). As the figure shows, each of the subtasks $T_{i+1}, \ldots, T_{i+j'-1}$ and $U_{j+1}, \ldots, U_{j+j'-1}$ has a window of length two. In addition, because $T \notin S_u$, $w(T_{i+j'})$ is either a 2-window starting at slot u or a 3-window starting at slot $u-1$; however, if $w(T_{i+j'})$ is a 2-window starting at slot u, then T has a group deadline at $u-1$ rather than u. We conclude that $w(T_{i+j'})$ is a 3-window and $T_{i+j'}$ is scheduled in slot $t' + j' + 1$.

Our strategy now is to identify another task that can be used as an intermediate for swapping. Because $\{T, U\} \subseteq S_{u-1}$ and $\{T, U\} \not\subseteq S_u$ and all processors are fully utilized, there exists a task V such that $V \in S_u$ and $V \notin S_{u-1}$. Let $k = \mathit{subtask}(V, u)$.

If $r(V_k) < u$, then the swapping shown in Figure 7(b) gives the required schedule. Also, if $r(V_k) = u \wedge V \notin S_{u+1}$, then the swapping in Figure 7(c) is valid. In the rest of the proof, we assume

$$r(V_k) = u \wedge V \in S_{u+1}.$$

Note that $V \in S_{u+1}$ implies that $d(V_k) = u + 1$, i.e., $|w(V_k)| = 2$. Therefore, by (P4), V is heavy. Consider the group deadline of V_k, $D(V_k)$. Let v be the earliest slot after u such that $V \notin S_v$. Note that

$$v \leq D(V_k). \tag{20}$$

(See Figure 7(d).) Let $V_{k+i'}$ be the subtask of V that is scheduled in slot $v-1$. If either $v < D(T_{i+j'})$ or $v = D(T_{i+j'}) \wedge b(T_{i+j'+i'}) = 0$, then $T_{i+j'+i'}$ is scheduled in slot v, and the swapping shown in Figure 7(d) is valid. In the rest of the proof, we assume that neither of these conditions holds, i.e.,

$$[D(T_{i+j'}) < v] \vee [v = D(T_{i+j'}) \wedge b(T_{i+j'+i'}) = 1]. \tag{21}$$

We claim that $u-1$ is a group deadline of V. As seen in Figure 7(d), V_{k-1} is not scheduled in slot $u-1$. Because $r(V_k) = u$, $d(V_{k-1})$ is either $u-1$ or u. If $d(V_{k-1}) = u-1$, then $b(V_{k-1}) = 0$ and $w(V_{k-1})$ is the final window of a job of V, and thus $u-1$ is a group deadline. If $d(V_{k-1}) = u$, then we reason as follows. Because V is a heavy task, by (P2) and (P4), $|w(V_{k-1})| \leq 3$. Because V_{k-1} is not scheduled in slots $u-1$ or u, $r(V_{k-1})$ has to be $u-2$. This implies that $|w(V_{k-1})| = 3$. Therefore, $u-1$ is a group deadline of V.

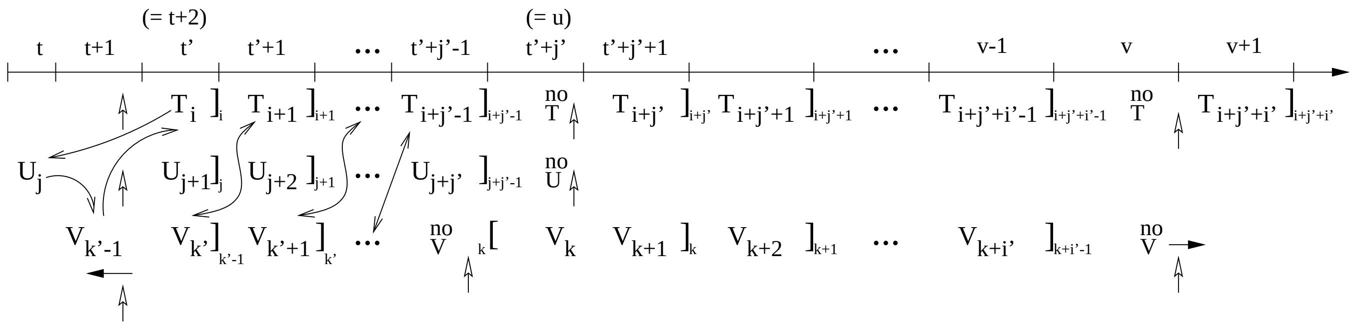


Figure 8: Subcase 4.A. $t' = t + 2$.

Having shown that $u-1$ is a group deadline of V, we now show that $pred(V, u-1) \leq pred(T, u)$. T has consecutive group deadlines at u and $succ(T, u) = D(T_{i+j'})$. Therefore, by (P5), the difference between u and $pred(T, u)$ is at most one more than $D(T_{i+j'}) - u$, i.e., $u - pred(T, u) \leq D(T_{i+j'}) - u + 1$. Therefore,

$$pred(T, u) \geq 2u - D(T_{i+j'}) - 1. \tag{22}$$

V has consecutive group deadlines at $u-1$ and $succ(V, u-1) = D(V_k)$. Hence, by (P5), the difference between $u-1$ and $pred(V, u-1)$ is at least one less than $D(V_k) - (u-1)$, i.e., $u - 1 - pred(V, u-1) \geq D(V_k) - u$. Thus,

$$pred(V, u-1) \leq 2u - D(V_k) - 1. \tag{23}$$

By (20), (21), (22), and (23), $pred(V, u-1) \leq 2u - D(V_k) - 1 \leq 2u - D(T_{i+j'}) - 1 \leq pred(T, u)$. Thus, $pred(V, u-1) \leq pred(T, u)$. Note also that $pred(V, u-1) = pred(T, u)$ if and only if $D(T_{i+j'}) = D(V_k) = v$. In addition, as seen in Figure 7(d), T cannot have a group deadline in the interval $[t', u-1]$. Therefore, we have the following.

$$pred(V, u-1) \leq pred(T, u) \leq t' - 1 \tag{24}$$

$$(pred(V, u-1) = pred(T, u)) \;\Rightarrow\; (D(T_{i+j'}) = v \;\wedge\; D(V_k) = v) \tag{25}$$

Recall that t' is either $t+2$ or $t+1$. We consider these two subcases next.

Subcase 4.A: $t' = t + 2$. In this case, we show that the swapping in Figure 8 is valid. To begin, note that $t' = t + 2$ implies that $T \notin S_{t+1}$ and $U \notin S_{t+1}$. Let $k' = k - j' + 1$. As Figure 8 shows,

- $w(V_{k-1})$ is either a 2- or 3-window beginning in slot $u - 2 = t' + j' - 2$ (this is because $u-1$ is a group deadline of V);

- for each l in the range $k' \leq l < k-1$, $w(V_l)$ is a 2-window (this is because, by (24), $pred(V, u-1) < t'$);

- each of $V_{k'}, \ldots, V_{k-1}$ is scheduled in the first slot of its window (this can be seen by inducting from right to left, starting with V_{k-1}).

Because $V_{k'}$ is released at $t' = t+2$, $V_{k'-1}$ has a deadline at either $t+1$ or $t+2$. We now prove that $d(V_{k'-1})$ cannot equal $t+1$. Assume, to the contrary, that $d(V_{k'-1}) = t+1$. This implies that $b(V_{k'-1}) = 0$, i.e., $w(V_{k'-1})$ is the final window of its job. Thus,

$$pred(V, u-1) = t+1. \tag{26}$$

If V has multiple group deadlines per job, then by (P5) and (P6), the difference between $pred(V, u-1)$ and $u-1$ is at least the difference between $u-1$ and $succ(V, u-1)$, i.e., $succ(V, u-1) - (u-1) \le (u-1) - pred(V, u-1)$. If V has one group deadline per job, then clearly $succ(V, u-1) - (u-1) = (u-1) - pred(V, u-1)$. In either case, by (26),

$$succ(V, u-1) \le 2u - t - 3. \tag{27}$$

Because $t' = t+2$, $w(T_i)$ is a 3-window. Thus, $pred(T, u) = t' - 1 = t+1$. By (P5), the difference between $succ(T, u)$ and u is at least one less than $u - pred(T, u)$, i.e., $succ(T, u) - u \ge u - (t+1) - 1$. Because $succ(T, u) = D(T_{i+j'})$, this implies that $D(T_{i+j'}) \ge 2u - t - 2$. Thus, by (21), $v \ge 2u - t - 2$. Because $succ(V, u-1) = D(V_k)$, by (20), we have $succ(V, u-1) \ge v$. Thus, $succ(V, u-1) \ge 2u - t - 2$, which contradicts (27). Therefore, we conclude that $d(V_{k'-1})$ cannot be $t+1$. Thus, we have the following.

$$d(V_{k'-1}) = t+2$$

We now show that $V_{k'-1}$ is scheduled in slot $t+1$, which implies that the swapping in Figure 8 is valid. Assume, to the contrary, that $V_{k'-1}$ is *not* scheduled at $t+1$. We have established that V is heavy and $d(V_{k'-1}) = t+2$. Moreover, $V_{k'-1}$ is not scheduled in slot $t+1$ (by assumption) or in slot $t+2$ (because $V_{k'}$ is scheduled there). By (P2)-(P4), this implies that $w(V_{k'-1})$ is a 3-window, $r(V_{k'-1}) = t$, and $V_{k'-1}$ is scheduled in slot t. As seen in Figure 8, $d(V_{k'-1}) = d(U_j)$, $b(V_{k'-1}) = b(U_j)$, and $D(V_{k'-1}) < D(U_j)$. Therefore, $V_{k'-1}$ has lower priority than U_j at time t. Recall that A is the set of tasks selected for execution in slot t according to our priority definition. Also recall that $U \notin A$. Because $U \notin A$, we have $V \notin A$. Therefore, $V \in S_t - A$. This contradicts our choice of U as a task of lowest priority in $S_t - A$.

Subcase 4.B: $t' = t+1$. In this case, we show that one of the swappings in Figure 9 is valid. We begin by showing that

$$pred(V, u-1) = t. \tag{28}$$

As in Subcase 4.A, it is possible to show that

- each of the subtasks $V_{k'}, \ldots, V_{k-2}$ has a window of length two and is scheduled in the first slot of its window, and

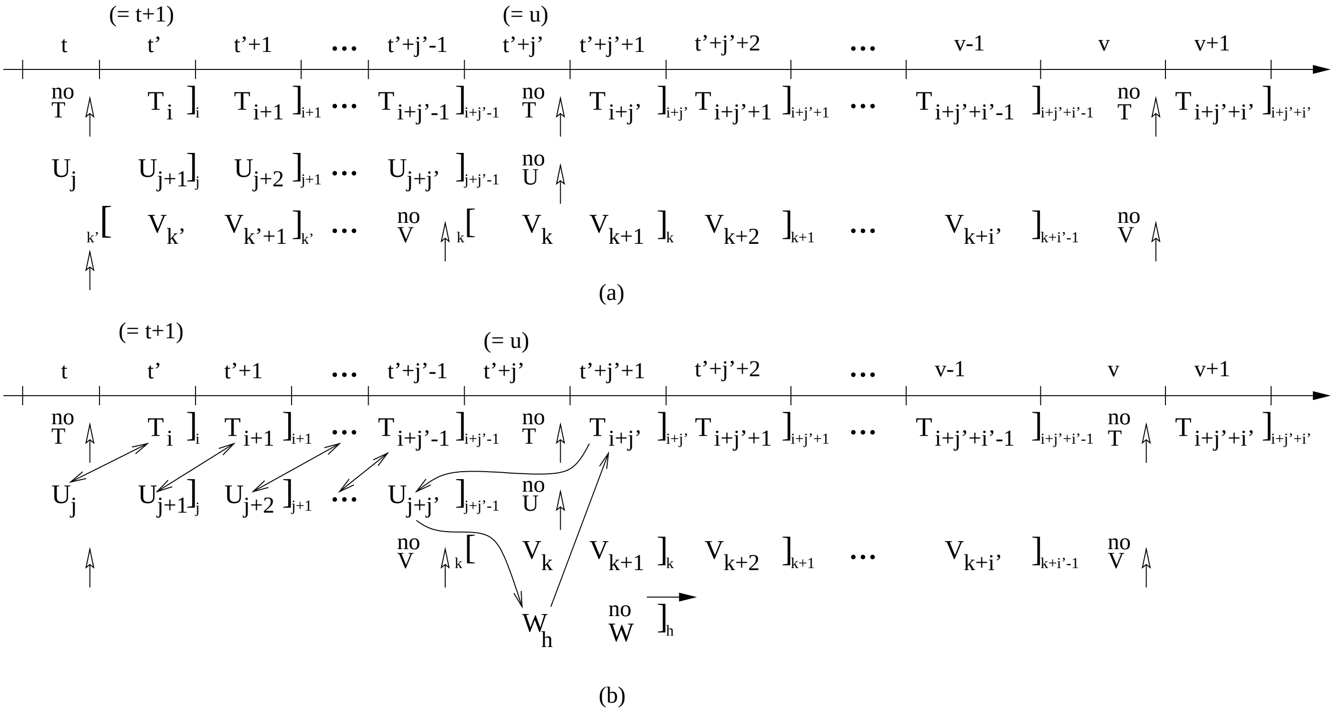


Figure 9: Subcase 4.B. In each inset, $t' = t+1$, $D(T_i) = D(U_j)$, and $D(V_k) = D(T_{i+j'})$. **(a)** $pred(V, u-1) = pred(T, u)$. **(b)** $d(W_h) > u$. (Continued on the following page.)

- V_{k-1} has a window of length two or three and is scheduled in the first slot of its window.

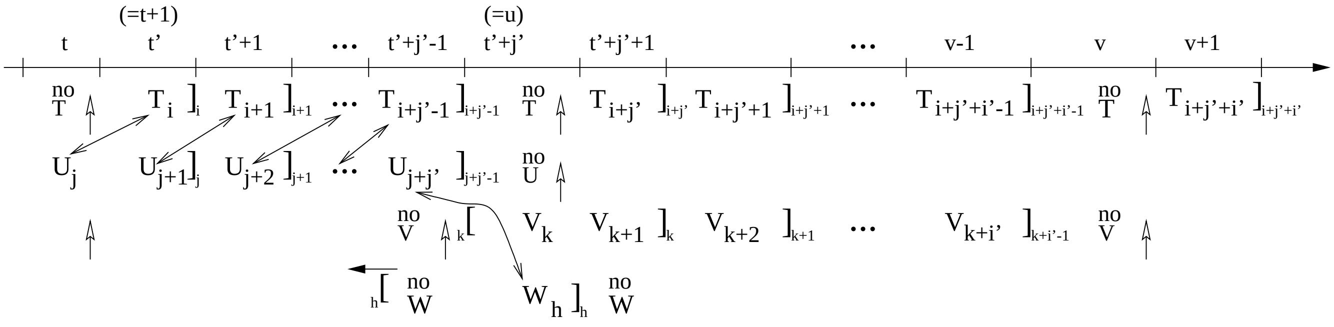
This is depicted in Figure 9(a). Because $r(V_{k'}) = t+1$, $d(V_{k'-1})$ is either t or $t+1$. If $d(V_{k'-1}) = t$, then $w(V_{k'-1})$ is the final window of its job. Therefore, $pred(V, u-1) = t$.

If $d(V_{k'-1}) = t+1$, then we reason as follows. Because V is heavy, by (P2) and (P4), $w(V_{k'-1})$ is of length two or three. If $w(V_{k'-1})$ is of length three, then clearly, $pred(V, u-1) = t$. Thus, it suffices to show tnat $w(V_{k'-1})$ is not of length two. Suppose, to the contrary, that $w(V_{k'-1}) = 2$. Because $d(V_{k'-1}) = t+1$, and because $V_{k'}$ is scheduled in slot $t+1$, $V_{k'-1}$ is scheduled in slot t. Observe that $d(V_{k'-1}) = t+1$, $d(U_j) = t+1$, $b(V_{k'-1}) = 1$, $b(U_j) = 1$, and $D(V_{k'-1}) < D(U_j)$. Thus, $V_{k'-1}$ has lower priority than U_j at t. This implies that $V \notin A$, i.e., $V_{k'-1} \in S_t - A$. However, this contradicts our choice of U as a task of lowest priority in $S_t - A$. This completes our proof of (28). By (24) and (28), we have the following.

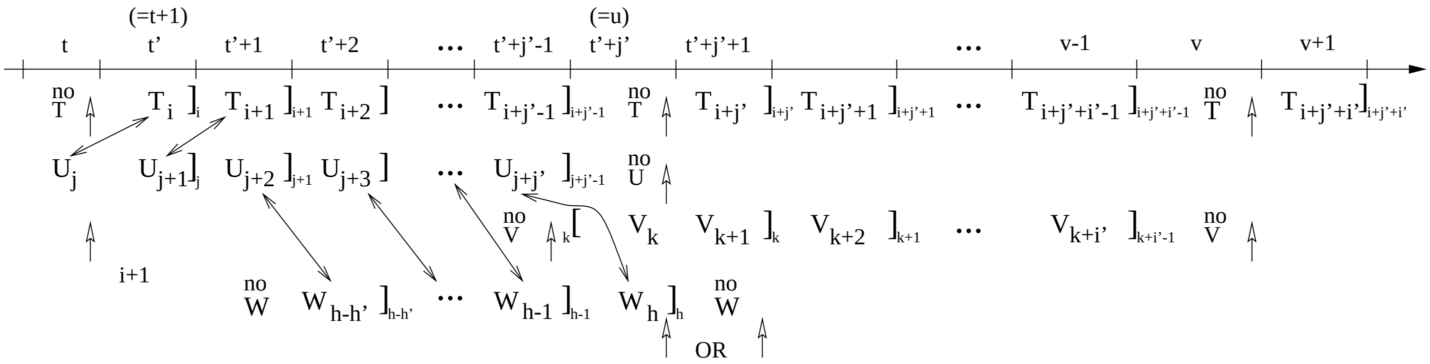
$$pred(T, u) = pred(V, u-1)$$

By (25), this implies that $D(T_{i+j'}) = D(V_k) = v$ (see Figure 9(a)).

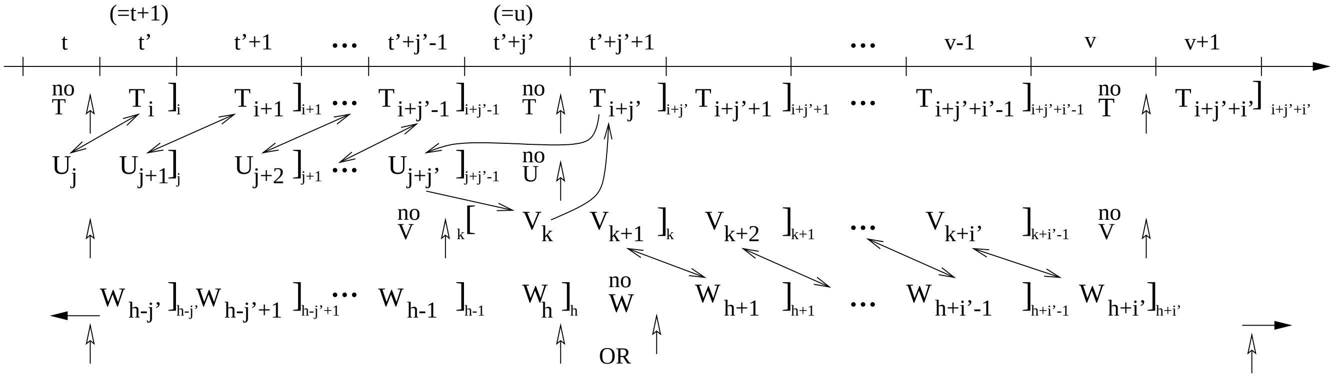
Because $T \notin S_u$ and $T \in S_{u+1}$, and because the processors are fully utilized, there exists a task W such that $W \in S_u$ and $W \notin S_{u+1}$. Let $h = subtask(W, u)$. If $d(W_h) > u$, then the swapping shown in Figure 9(b) is



(c)



(d)



(e)

Figure 9: (Continued) **(c)** $d(W_h) = u$ and $W \notin S_{u-1}$. **(d)** $d(W_h) = u$, $W \in S_{u-1}$, and $W \notin S_w$ for some w in $[t', u-1]$. **(e)** $d(W_h) = u$, $W \in S_{u-1}$, and W's most recent group deadline before the one at u or $u+1$ is at or before t.

valid. In the rest of the proof, we assume that

$$d(W_h) = u.$$

In this case, we show that one of the swappings in Figure 9(c)-(e) are valid. If $W \notin S_{u-1}$, then the swapping shown in Figure 9(c) is valid. In the rest of the proof, we assume

$$W \in S_{u-1}.$$

In this case, we have $r(W_h) = u - 1$, i.e., $|w(W_{h-1})| = 2$. Thus, by (P4), W is heavy.

We now show that W has a group deadline at time u or $u+1$ (refer to Figure 9(d)). Because $W_{h+1} \notin S_{u+1}$ and W is heavy, by (P2) and (P4), W_{h+1} has to be scheduled at time $u+2$, and $w(W_{h+1})$ is either a 3-window beginning at slot u or a 2-window beginning at slot $u+1$. In the former case, $u+1$ is the middle slot of a 3-window, and hence a group deadline of W. In the latter case, u is the final slot of a job of W, and hence a group deadline of W.

We now look at earlier subtasks of W. If there exists w such that $t' \leq w \leq u-1$ and $W \notin S_w$, then a swapping similar to that shown in Figure 9(d) is valid and produces the desired schedule. In the rest of the proof, we assume that for each w in the range $t' \leq w \leq u$, $W \in S_w$. This implies that, at each slot in the interval $[t', u]$, a subtask of W is scheduled in the last slot of its window (recall that W is heavy). This is illustrated in Figure 9(e). As seen in the figure, each of the subtasks $W_{h-j'+1}, \ldots, W_h$ has a window of length two. This implies that the most recent group deadline of W before the one at u or $u+1$ occurs at or before time t, i.e.,

$$\begin{aligned} (u \text{ is a group deadline of } W \quad &\Rightarrow \quad \mathit{pred}(W, u) \leq t) \ \wedge \\ (u+1 \text{ is a group deadline of } W \quad &\Rightarrow \quad \mathit{pred}(W, u+1) \leq t). \end{aligned} \tag{29}$$

We now show that W's next group deadline after the one at u or $u+1$ occurs after time v, which implies that the swapping shown in Figure 9(e) is valid. There are two possibilities to consider, depending on whether W has a group deadline at u or $u+1$. In both cases, by (20), (23), and (28), we have

$$v \leq 2u - t - 1. \tag{30}$$

u is a group deadline of W. In this case, W_h is the last subtask of its job. By (P5) and (P6), the difference between $\mathit{succ}(W, u)$ and u is at least the difference between u and $\mathit{pred}(W, u)$, i.e., $\mathit{succ}(W, u) - u \geq u - \mathit{pred}(W, u)$. Furthermore, by (29), $\mathit{pred}(W, u) \leq t$. Therefore, $\mathit{succ}(W, u) \geq 2u - t$. By (30), this implies that $\mathit{succ}(W, u) > v$.

$u+1$ is a group deadline of W. In this case, by (P5), the difference between $\mathit{succ}(W, u+1)$ and $u+1$ is at least one less than the difference between $u+1$ and $\mathit{pred}(W, u+1)$, i.e., $\mathit{succ}(W, u+1) - (u+1) \geq (u+1) - \mathit{pred}(W, u+1) - 1$. Therefore, $\mathit{succ}(W, u+1) \geq 2u - \mathit{pred}(W, u+1) + 1$. Furthermore, by (29), $\mathit{pred}(W, u+1) \leq t$. This implies that $\mathit{succ}(W, u+1) \geq 2u - t + 1$. Therefore, by (30), $\mathit{succ}(W, u+1) > v$.

This exhausts all the possibilities if T and U are both heavy, and concludes the proof of Lemma 2. $\square$

Theorem 1 *Our algorithm generates a Pfair schedule for any feasible task set.*

Proof: If a task set τ is feasible, then it has a Pfair schedule S. By repeatedly applying Lemma 2, as necessary, first at time slot 0, then at time slot 1, etc., we can show the existence of Pfair schedules $S = S^0, S^1, \ldots, S^{L-1}, S^L$, where where L is the least common multiple of $\{T.p \mid T \in \tau\}$, such that, in each S^t, the scheduling decisions within the first t time units are in accordance with our priority definition. The existence of schedule S^L shows that a Pfair schedule can be produced for τ by scheduling tasks according to our priority definition. $\square$

5 Two-Processor Systems

In this section, we prove that no tie-breaking information is needed in two-processor systems, i.e., pseudo-deadlines alone suffice. The proof makes use of the following two properties, which are proved in Appendix A.

(P7) Let S be a Pfair schedule for $M = 2$ processors. Suppose that $T_i \in S_t$, $U_j \in S_t$, and $d(U_j) > t$. Then, there exists a subtask V_k scheduled in S after slot t such that $r(V_k) \leq t$. Informally, if U_j is "right-movable" out of slot t, then some V_k must be "left-movable" into t.

(P8) Let S be a Pfair schedule for $M = 2$ processors. Consider time slots t and u, where $t < u$. Let A be the set of all subtasks scheduled by S in $[t, u]$. Suppose that there exists some task W that has no subtask in A. Then, there exists a subtask $T_i \in A$ such that $r(T_i) < t$ or $d(T_i) > u$.

Theorem 2 *If M, the number of processors, is two, then no tie-breaking information is needed.*

Proof: As in Lemma 2, the "difficult" case depicted in Figure 10(a) is the main problem: we wish to swap U_j and T_i, where U_j is scheduled in slot t and T_i is scheduled in slot $t' > t$, but U_{j+1} is scheduled in slot t'. This case can arise only if $d(T_i) = d(U_j) = t'$.

Because U_{j+1} is scheduled in slot t' and $d(U_j) = t'$, we have $d(U_{j+1}) > t'$ — informally, U_{j+1} is "right-movable" out of slot t'. Thus, by (P7), there exists a subtask released at or before t' that is scheduled after t' — informally, such a subtask is "left-movable" into slot t'. Let W_k be the earliest such "left-movable" subtask. Assume that W_k is scheduled in slot w, where $w > t'$, as shown in Figure 10(a) (note that W_k could be T_{i+1}).

We would like to swap W_k with U_{j+1}. However, we may not be able to directly swap W_k and U_{j+1} because slot w may be after U_{j+1}'s deadline. If W_k and U_{j+1} are not directly swappable, then we instead try to inductively move W_k to the left by swapping it with other subtasks, one step at a time.

Suppose we have managed to move W_k from slot w to slot v, as shown in Figure 10(b). We claim that if W_k and U_{j+1} are not directly swappable, then there *must* be a subtask scheduled within $[t'+1, v-1]$ that can be swapped with W_k. To see this, assume that none of the subtasks scheduled in $[t'+1, v-1]$ can be swapped with W_k (see Figure 10(c)). Then, each of these subtasks has a deadline before v. Also, because W_k was chosen

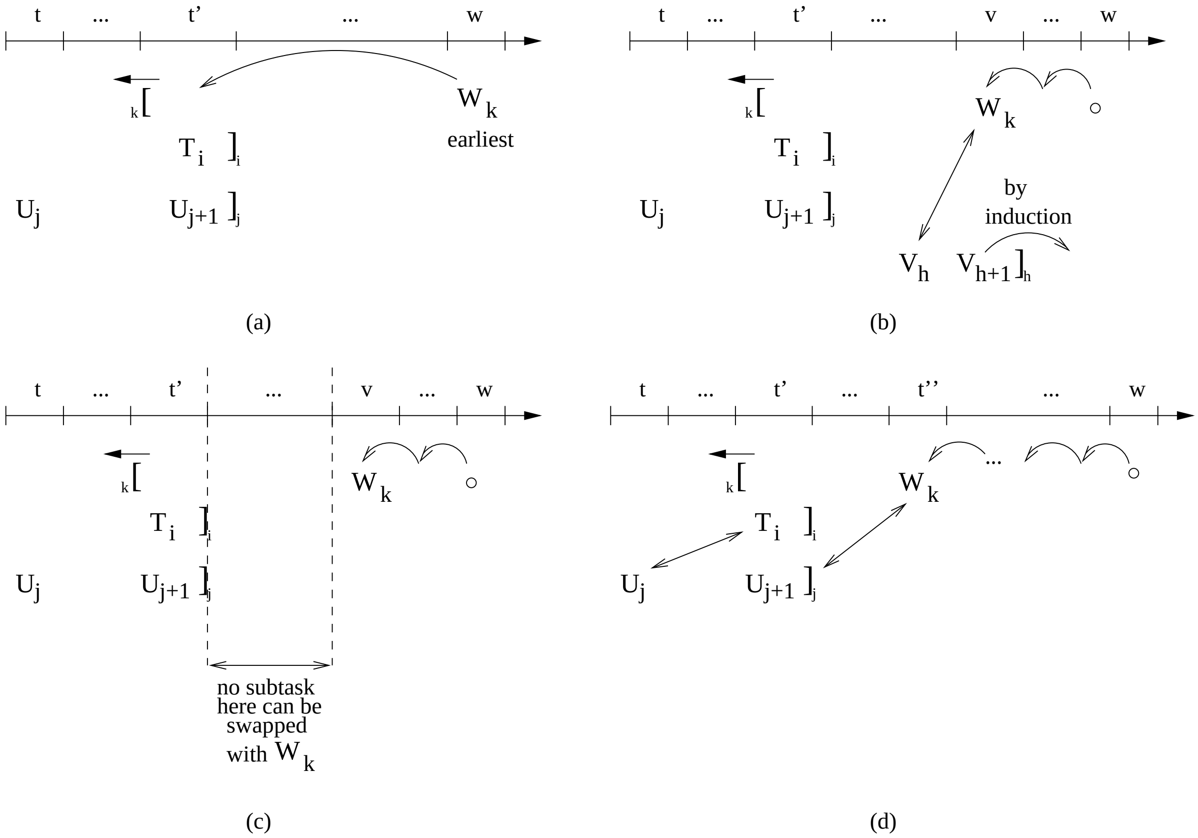


Figure 10: Two-processor systems. In each inset, $d(U_j) = d(T_i)$ and $r(W_k) \leq t'$. **(a)** A depiction of the "difficult" case. **(b)** W_k is scheduled in slot v, and $d(V_h) \geq v$. **(c)** Each task in $[t'+1, v-1]$ is released in $[t'+1, v-1]$ and has a deadline in $[t'+1, v-1]$. **(d)** W_k is scheduled in slot t'' and can be swapped with U_{j+1}.

as the earliest subtask such that $r(W_k) \leq t'$, each subtask in $[t'+1, v-1]$ is released at or after $t'+1$. Clearly, W has no subtask scheduled within $[t'+1, v-1]$. Thus, we have a contradiction of (P8).

We conclude that if W_k and U_{j+1} are not directly swappable, then there exists some subtask V_h scheduled in $[t'+1, v-1]$ that can be swapped with W_k, i.e., V_h's deadline is at or after v. If V_{h+1} is not scheduled in slot v, then swapping V_h and W_k is easy. Unfortunately, if V_{h+1} *is* scheduled in slot v, as depicted in Figure 10(b), then we seem to have a problem. However, this problem is exactly of the same form as the original one we were confronted with, i.e., this problem is exactly like the situation illustrated in Figure 10(a). We can therefore recursively apply the procedure described here to move V_{h+1} out of slot v. This recursive process cannot extend past slot L, where L is the least common multiple of $\{T.p \mid T \in \tau\}$, so it will eventually terminate.

It follows that we can eventually swap U_{j+1} and W_k directly, as depicted in Figure 10(d) (if U_{j+2} is scheduled in the same slot as W_k, then the above process once again must be applied recursively). Once U_{j+1} and W_k have been swapped, U_j and T_i — the two subtasks we originally wanted to swap — can be swapped. □

A similar argument shows that no tie-breaking information is needed on one processor, but this was previously known [3]. Looking at Figure 10, we can see why two-processor systems are different from systems of three or more processors. To move subtask U_{j+1} to the right out of slot t', some subtask W_k *must* be moved to the left into slot t'. If W_{k-1} is scheduled in slot t', then moving W_k into slot t' potentially causes a new problem that must be addressed. However, this can happen on a two-processor system only if W_k is T_{i+1}. The fact that T_i is scheduled in slot t' is not a problem, because this is the very subtask we wish to move to an earlier slot.

6 Counterexamples

According to our priority definition, each task T is prioritized at time t by the triple $(d(T_i), b(T_i), D(T_i))$, where $subtask(T,t) = i$. In this section, we present a collection of counterexamples that show that this priority definition cannot be substantially simplified. We begin by considering b.

Theorem 3 *If our priority definition is changed by eliminating b, then there exists a feasible task set that is not correctly scheduled.*

Proof: In each proof in this section, an example task set is considered that fully utilizes a system of M processors for some M. Each such task set consists of a set A of tasks of one weight and a set B of tasks of another weight. We show that if this task set is scheduled with the newly-proposed priority definition, then a time slot is reached at which fewer than M tasks are scheduled. This implies that more than M tasks are scheduled at some future time slot, which is a contradiction. In the proof of Theorem 3, we explain the contradictory schedule in detail. The subsequent proofs in this section are sketched more briefly.

Consider a task set consisting of a set A of eight light tasks with weight $1/3$ and a set B of three light tasks with weight $4/9$. Total utilization is four, so we should be able to schedule this task set on four processors. Consider the schedule shown in Figure 11(a). In this figure, tasks of a given weight are shown together. Each window is shown on a separate line and is depicted by showing the time slots it spans. Each column corresponds to a time slot. A slot t within a window is denoted by either an integer value or a dash. An integer value n means that n of the subtasks that must execute within that window are scheduled in slot t. A dash means either that no such subtask is scheduled in slot t, or the schedule at slot t is being left unspecified — in this and subsequent figures, we show only enough of the schedule being considered to derive a contradiction.

As seen in Figure 11(a), each job of a task with weight $1/3$ consists of one three-slot window. Each job of a task with weight $4/9$ consists of four three-slot windows, with consecutive windows overlapping by one slot. The first subtask of each task has a pseudo-deadline at slot 2. Because b has been eliminated, we can break this tie arbitrarily. In slot 0, we schedule four of the tasks from set A. In slot 1, we schedule the other four tasks from set A. In slot 2, the three tasks from set B are the only tasks with subtasks that are eligible for execution. This means that we can only schedule three subtasks in slot 2, which is a contradiction.

Although the counterexample given here involves only light tasks, other counterexamples exist that involve either heavy tasks only or a mixture of light and heavy tasks. □

Our definition of $D(T_i)$ ensures that if T is light and U is heavy and if $subtask(T,t) = i \wedge subtask(U,t) = j \wedge d(T_i) = d(U_j) \wedge b(T_i) = b(U_j)$, then U has higher priority. The following theorem shows that it is necessary

```
slot number:  0  1  2  3  4  5  6  7  8
   A(8x1/3):  4  4  _
   B(3x4/9):  _  _  3
                    _  _  _
                          _  _  _
                                _  _  _
```

(a)

```
slot number:  0  1  2  3  4  5  6  7  8  9  10
  A(5x5/11):  2  3  _
                    2  3  _
                          2  2  1
 B(2x19/22):  2  _              1  4  _
                 1  1                2  2  1
                    1  1
                       _  2
                          _  2
                             _  2
                                _  _  2
                                      _  2
                                         _  2 ...
```

(b)

```
slot number:  0  1  2  3  4  5  6
   A(3x5/7):  3  _
                 3  _
                    2  1  _
 B(2x13/14):  1  1        _  _
                 _  2        _  _
                    _  2
                       _  _
                          _  _
                             _  _ ...
```

(c)

```
slot number:  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
   A(3x8/9):  3  _
                 3  _
                    3  _
B(10x14/15):  9  1     3  _
                 8  2     3  _
                    7  3     3  _
                       6  4     3  _
                          5  5     2  1
                             4  6        _  _
                                3  7        _  _
                                   3  7        _  _
                                      3  _        _  _
                                         _  _        _  _
                                            _  _
                                               _  _
                                                  _  _
                                                     _  _
```

(d)

```
slot number:  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
   A(3x8/9):  2  1
                 1  2
                    _  3
B(10x14/15):  10 _     _  3
                 10 _     _  3
                    10 _     _  3
                       9  1     _  3
                          8  2     _  3
                             7  3        2  1
                                6  4        1  2
                                   5  5        _  3
                                      4  6        _  3
                                         4  6        _  3
                                            4  6
                                               4  6
                                                  3  7
                                                     2  8
```

(e)

```
slot number:  0  1  2  3  4  5  6  7  8
   A(9x7/9):  9  _
                 9  _
                    9  _
  B(12x5/6):  8  4     5  4  _
                 4  8        _  _
                    _  12       _  _
                       _  12       _  _
                          _  _
```

(f)

```
slot number:  0  1  2  3
   A(3x1/2):  3  _
   B(2x3/4):  _  2
                 _  _
                    _  _
```

(g)

Figure 11: Counterexamples. **(a)** Theorem 3. **(b)** Theorem 4. **(c)** Theorems 5 and 8. **(d)** and **(e)** Theorem 6. **(f)** Theorem 7. **(g)** Theorem 9.

to tie-break such a situation in favor of the heavy task.

Theorem 4 *Let us change the definition of D as follows: if T is light, then $D(T_i)$ is a randomly-selected value. Then, there exists a feasible task set that is not correctly scheduled.*

Proof: Consider a task set consisting of a set A of five light tasks with weight $5/11$ and a set B of two heavy tasks with weight $19/22$. Total utilization is four, so we should be able to schedule this task set on four processors. Consider the schedule shown in Figure 11(b), which is possible given the proposed priority definition. In time slot 10, there are only three subtasks that are eligible for execution. Contradiction. □

The previous counterexamples give rise to the possibility that $D(T_i)$ is actually only needed to tie-break heavy tasks over light tasks. The next theorem shows that this is not the case.

Theorem 5 *Let us change the definition of D as follows: if T is heavy, then $D(T_i)$ is one (if T is light, then $D(T_i)$ is zero as before). Then, there exists a feasible task set that is not correctly scheduled.*

Proof: Consider a task set, to be scheduled on four processors, consisting of a set A of three heavy tasks with weight $5/7$ and a set B of two heavy tasks with weight $13/14$. The proposed priority definition allows the schedule shown in Figure 11(c). Note that only three subtasks are eligible at time slot 3. Contradiction. □

Given the previous counterexample, one may wonder if the definition of D can be weakened so that ties among heavy tasks are statically resolved. The following theorem shows that this is unlikely.

Theorem 6 *If D is changed so that ties among heavy tasks are statically broken by weight (as before, D is defined to favor heavy tasks over light tasks), then there exists a feasible task set that is not correctly scheduled.*

Proof: Consider a task set, to be scheduled on 12 processors, consisting of a set A of three heavy tasks with weight $8/9$ and a set B of ten heavy tasks with weight $14/15$. First, suppose that D is defined to statically tie-break the set-A tasks over the set-B tasks. Then, the schedule shown in Figure 11(d) is possible. In this schedule, only 11 subtasks are eligible at time slot 8, which is a contradiction. Second, suppose that D is defined to statically tie-break the set-B tasks over the set-A tasks. In this case, the schedule shown in Figure 11(e) is possible. In this schedule, only 11 subtasks are eligible at time slot 14. Once again, we have a contradiction. □

From the previous theorem, it follows that D almost certainly must be defined to dynamically tie-break heavy tasks. (Note, for example, that Theorem 6 leaves open the possibility of statically defining D so that some set-A tasks are favored over set-B tasks, but other set-A tasks are not favored over set-B tasks.) One obvious approach to try that is "less dynamic" than ours is to define $D(T_i)$ based on the time of the next job release of task T, which gives the deadline of the current job release of T. The next two theorems show that using job deadlines does not work; in the first of these theorems, farther job deadlines are given higher priority, and in the second, nearer job deadlines are given higher priority.

Theorem 7 *Let us change the definition of D as follows: if T is heavy, then $D(T_i)$ is the time of the next job release of task T. Then, there exists a feasible task set that is not correctly scheduled.*

Proof: Consider a task set, to be scheduled on 17 processors, consisting of a set A of nine heavy tasks with weight $7/9$ and a set B of 12 heavy tasks with weight $5/6$. The proposed priority definition allows the schedule shown in Figure 11(f) (note that the newly-proposed definition of D favors set-A tasks over set-B tasks). In this schedule, only 16 subtasks are eligible at time slot 4. Contradiction. □

Theorem 8 *Let us change the definition of D as follows: if T is heavy, then $D(T_i)$ is $1/t$, where t is the time of the next job release of task T. Then, there exists a feasible task set that is not correctly scheduled.*

Proof: A contradiction is reached by using the task set and schedule shown in Figure 11(c), which was used previously in the proof of Theorem 5 (note that the newly-proposed definition of D favors set-A tasks). □

In Section 5, we showed that neither b nor D is needed in two-processor systems. We do not know if both are needed on all systems of three or more processors, but we do know that at least one is needed in any such system. This is shown next. (Note that Theorems 3 through 5 and 8 apply on systems of four or more processors.)

Theorem 9 *If our priority definition is changed by eliminating both b and D, then there exists a task set that is feasible on three processors that is not correctly scheduled.*

Proof: Consider a task set, to be scheduled on three processors, consisting of a set A of three heavy tasks with weight $1/2$ and a set B of two heavy tasks with weight $3/4$. The proposed priority definition allows the schedule shown in Figure 11(g). In this schedule, only two subtasks are eligible at time slot 1. Contradiction. (Note that *either b or D* would correctly tie-break these tasks.) □

7 Concluding Remarks

We have shown that in a Pfair-scheduled multiprocessor system, each task can be prioritized using a pseudo-deadline and only two tie-break parameters. We have also presented a collection of counterexamples that shows that an even simpler priority definition is unlikely. For the special case of a two-processor system, we have shown that *no* tie-break parameters are needed. In proving that our priority definition suffices, we have used an inductive swapping argument in which any arbitrary Pfair schedule is converted into one allowed by our priority definition. This proof reveals many properties fundamental to Pfair scheduling. Indeed, we view this proof as one of the most important contributions of this paper because it gives researchers interested in Pfair-scheduled systems a new set of techniques that can be applied to reason about such systems.

We ourselves have been able to extend the arguments given in this paper to show that our simplified PD algorithm can be applied to optimally schedule sporadic tasks [2]. In addition, we have shown that our algorithm can be applied within a model that allows subtasks to sometimes execute prior to their windows [1]: in this model, if T_i and T_{i+1} are part of the same job, then T_{i+1} becomes eligible for execution immediately after T_i executes. This model is attractive because less bookkeeping information is required at run-time for determining when a subtask is eligible for execution. The results in [1] and [2] were obtained by layering some additional reasoning on top of the swapping proof given in this paper. In both cases, very few changes to the swapping proof were required. We believe that such results would have been very difficult to obtain using the proof techniques of the original PD paper [5].

Acknowledgement: We are grateful to Sanjoy Baruah, Mark Moir, and Srikanth Ramamurthy for many helpful discussion on the subject of this paper.

References

[1] J. Anderson and A. Srinivasan. Early release pfair scheduling. Unpublished manuscript.

[2] J. Anderson and A. Srinivasan. Pfair scheduling of sporadic tasks. Unpublished manuscript.

[3] S. Baruah. Fairness in periodic real-time scheduling. In *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pages 200–209, December 1995.

[4] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[5] S. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the the 9th International Parallel Processing Symposium*, pages 280–288, April 1995.

[6] S. Baruah, J. Gehrke, C.G. Plaxton, I. Stoica, H. Abdel-Wahab, and K. Jeffay. Fair on-line scheduling of a dynamic set of tasks on a single resource. *Information Processing Letters*, 64(1):43–51, October 1997.

[7] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–315, 1978.

Appendix A: Proof of Properties (P1) through (P8)

In this appendix, we prove properties (P1)-(P8). As before, we restrict attention to a task set that fully utilizes the available M processors, i.e., $\sum_{T\in\tau}\frac{T.e}{T.p}=M$. Recall that $T.e$ and $T.p$ are assumed to be relatively prime for each task T. Throughout this appendix, we let S denote an arbitrary Pfair schedule.

Before proving (P1), we state a simple property that indicates how a task's lag changes from slot to slot. This property follows directly from the definition of lag given in (1).

Claim 1: Let U be any task and let $t \geq 0$. Then,

$$lag(U,t+1)=\begin{cases} lag(U,t)+U.e/U.p, & \text{if } U\notin S_t \\ lag(U,t)+U.e/U.p-1, & \text{if } U\in S_t. \end{cases} \qquad \square$$

(P1) through (P6) pertain to just a single task. For brevity, we let T denote this task and abbreviate $T.e$ and $T.p$ as e and p, respectively.

(P1) The windows of each task T are symmetric within each job of T, i.e., $|w(T_{ke+i})| = |w(T_{ke+e+1-i})|$, where $1 \leq i \leq e$, and $k \geq 0$.

Proof: As explained in Section 2, $lag(T,t)=0$ for $t=0,p,2p,3p,\ldots$. This implies that the windows of each job of T are exactly the same. Thus, we can prove (P1) by simply considering the first job of T, i.e., by showing that $|w(T_i)| = |w(T_{e+1-i})|$ holds, where $1 \leq i \leq e$. Call this job J. The proof proceeds as follows. We first obtain a formula for the length of each window of J. Using this formula, we establish that the length of the i^{th} window of J is equal to the length of the $(e+1-i)^{th}$ window of J.

We begin by deriving an expression that gives the last slot of each window of J (i.e., the deadline of the corresponding subtask). Let u_i be the total number of time slots up to and including the last slot of the i^{th} window, where $1 \leq i \leq e$. Note that u_i is the earliest slot t such that, if $i-1$ units of computation of T have been previously scheduled, but the i^{th} unit of computation of T has not been previously scheduled, then *not* scheduling the i^{th} unit of computation in slot t would result in T's lag exceeding one. By Claim 1, this implies that u_i is the minimum t such that $t\frac{e}{p}-(i-1)\geq 1-\frac{e}{p}$. (We get t instead of $t-1$ in this expression because slots are numbered starting at 0. Thus, slot t is the $(t+1)^{st}$ slot, i.e., there are t previous slots.) Solving yields

$$u_i=\begin{cases} \lfloor\frac{ip}{e}\rfloor, & \text{if } 1\leq i<e \\ p-1, & \text{if } i=e, \end{cases}$$

or equivalently,

$$u_i=\left\lceil\frac{ip}{e}\right\rceil-1.$$

Note that this last expression expression for u_i matches Equation (5) given in Section 3 for $d(T_i)$.

We now derive an expression that gives the first slot of each window of J (i.e., the slot at which the corresponding subtask is released). Let t_i be the total number of time slots up to and including the first slot of the i^{th} window, where $1 \leq i \leq e$. Then, t_i is the earliest slot t such that, if $i-1$ units of computation of T have been previously scheduled, but the i^{th} unit of computation of T has not been previously scheduled, then scheduling the i^{th} unit of computation in slot t would not cause T's lag to be less than -1. By Claim 1, this implies that t_i is the minimum t such that $t\frac{e}{p} - (i-1) > -\frac{e}{p}$. Solving yields

$$t_i = \left\lfloor \frac{(i-1)p}{e} \right\rfloor$$

Notice that this expression for t_i matches Equation (4) given in Section 3 for $r(T_i)$. By definition, $|w(T_i)| = u_i + 1 - t_i$. Therefore,

$$|w(T_i)| = \left\lceil \frac{ip}{e} \right\rceil - \left\lfloor \frac{(i-1)p}{e} \right\rfloor . \tag{31}$$

We are now in a position to prove (P1).

$$\begin{aligned}
|w(T_{e+1-i})| &= \left\lceil \frac{(e+1-i)p}{e} \right\rceil - \left\lfloor \frac{(e-i)p}{e} \right\rfloor \\
&= \left(\left\lceil \frac{(1-i)p}{e} \right\rceil + p \right) - \left(\left\lfloor \frac{-ip}{e} \right\rfloor + p \right) \\
&= \left\lceil \frac{(1-i)p}{e} \right\rceil - \left\lfloor \frac{-ip}{e} \right\rfloor \\
&= \left\lceil \frac{-(i-1)p}{e} \right\rceil + \left\lceil \frac{ip}{e} \right\rceil \\
&= -\left\lfloor \frac{(i-1)p}{e} \right\rfloor + \left\lceil \frac{ip}{e} \right\rceil
\end{aligned}$$

Thus, $|w(T_i)| = |w(T_{e+1-i})|$. □

(P2) The length of each of task T's windows is either $\lceil \frac{p}{e} \rceil$ or $\lceil \frac{p}{e} \rceil + 1$.

Proof: As in the proof of (P1), we can limit attention to subtasks T_i, where $1 \leq i \leq e$, i.e., the subtasks of the first job of T. By (31), we have

$$\begin{aligned}
|w(T_i)| &= \left\lceil \frac{ip}{e} \right\rceil - \left\lfloor \frac{(i-1)p}{e} \right\rfloor \\
&= \left\lceil \frac{ip}{e} \right\rceil - \left\lfloor \frac{ip}{e} - \frac{p}{e} \right\rfloor \\
&= \left\lceil \frac{ip}{e} \right\rceil + \left\lceil \frac{p}{e} - \frac{ip}{e} \right\rceil .
\end{aligned}$$

It is easy to see that this last expression equals either $\lceil \frac{p}{e} \rceil$ or $\lceil \frac{p}{e} \rceil + 1$. □

(P3) The first window of each job of a task is a minimal window of that task.

Proof: By (31), $|w(T_1)| = \lceil \frac{p}{e} \rceil$. Thus, by (P2), $w(T_1)$ is a minimal window of T. As noted before, this implies that the first window of each job of a T is a minimal window of T. $\square$

(P4) A task has a 2-window if and only if it is heavy.

Proof: As shown above, $|w(T_1)| = \lceil \frac{p}{e} \rceil$. Note that $|w(T_1)| = 2$ if and only if $\frac{e}{p} \geq \frac{1}{2}$. Thus, a task is heavy if and only if its very first window (and hence, the first window of each of its jobs) is a 2-window. $\square$

(P5) If t and t' are successive group deadlines of a heavy task T, then $t' - t$ is either $\left\lfloor \frac{p}{p-e} \right\rfloor$ or $\left\lfloor \frac{p}{p-e} \right\rfloor + 1$.

Proof: As before, (P5) can be proved by considering just the first job J of T (taking the "last" group deadline before J to be at time -1). Suppose that each subtask of J is scheduled in the first slot of its window. Then, the slots that remain empty exactly correspond to the group deadlines of T. We can analytically determine which slots must be empty, and thereby deduce expressions for characterizing each of J's group deadlines. If J has just a single group deadline, then it must be the last slot of J, i.e., all group deadlines of T are job deadlines. It can be shown that this is the case if only if $e = p - 1$. Thus, the distance between any two consecutive group deadlines is exactly $p = \left\lfloor \frac{p}{p-e} \right\rfloor$. In the rest of the proof, we assume that J has multiple group deadlines, which implies that it has at least one 3-window.

Let t_1 be the middle slot of the first 3-window of J. Because each subtask of J is executed in its first slot, a subtask of T is scheduled in each slot prior to t_1, and no subtask of T is eligible to be scheduled in slot t_1. By Claim 1, this implies that t_1 is the minimum u satisfying

$$lag(T, u) = u(e/p - 1) \leq -e/p.$$

Thus, t_1 is the minimum u satisfying $u \geq e/(p - e)$. Because t_1 is an integer, we have $t_1 = \left\lceil \frac{e}{p-e} \right\rceil$. This gives us the position of J's first group deadline.

Suppose that J has a second 3-window. Let t_2 be the middle slot of this window. We can calculate an expression for t_2 as above. In this case, t_2 is the minimum u satisfying

$$lag(T, u) = (u - 1)(e/p - 1) + e/p \leq -e/p.$$

The above expression given for $lag(T, u)$ follows from Claim 1 and the fact that subtasks of T have been scheduled in all previous slots but one. Solving as before, we get $t_2 = \left\lceil \frac{e+p}{p-e} \right\rceil$. This gives us the position of J's second group deadline.

Generalizing this argument, and using the fact that J's last group deadline is at slot $p - 1$, we can conclude that J has $p - e$ group deadlines, $t_1, \ldots, t_{p-e}$, where

$$t_j = \left\lceil \frac{e + (j-1)p}{p - e} \right\rceil. \tag{32}$$

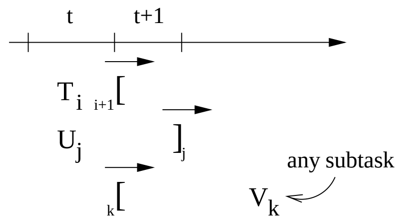


Figure 12: Proof of P7. $r(T_{i+1}) > t$, $d(U_j) > t$, and each subtask scheduled after t is released after t.

Now, consider the following.

$$\begin{aligned} t_{j+1} - t_j &= \left\lceil \frac{e + jp}{p - e} \right\rceil - \left\lceil \frac{e + (j-1)p}{p - e} \right\rceil \\ &= \left\lceil \frac{e + jp}{p - e} \right\rceil - \left\lceil \frac{e + jp}{p - e} - \frac{p}{p - e} \right\rceil \\ &= \left\lceil \frac{e + jp}{p - e} \right\rceil + \left\lfloor \frac{p}{p - e} - \frac{e + jp}{p - e} \right\rfloor \end{aligned}$$

It is straightforward to show that this last expression equals either $\left\lfloor \frac{p}{p-e} \right\rfloor$ or $\left\lfloor \frac{p}{p-e} \right\rfloor + 1$. □

(P6) Let T be a heavy task with more than one group deadline per job. Let t and t' (respectively, u and u') be successive group deadlines of T, where t' (respectively, u') is the first (respectively, last) group deadline within a job of T (for the first job of T, take t to be -1). Then, $t' - t = u' - u + 1$.

Proof: By (P1), the windows of each job J of T are symmetric. The reason why $t' - t$ is one greater than $u' - u$ is that, while the last slot of T is a group deadline, the first slot of T is not. Thus, t' is the middle slot of the first 3-window of J and t is the last slot of the job of T before J (or -1 if J is the first job of T). In contrast, u' is the last slot of J, and u is the middle slot of the last 3-window of J. □

(P7) Let S be a Pfair schedule for $M = 2$ processors. Suppose that $T_i \in S_t$, $U_j \in S_t$, and $d(U_j) > t$. Then, there exists a subtask V_k scheduled in S after slot t such that $r(V_k) \leq t$. Informally, if U_j is "right-movable" out of slot t, then some V_k must be "left-movable" into t.

Proof: Because $T_i \in S_t$, we have $r(T_{i+1}) \geq t$. If $r(T_{i+1}) = t$, then (P7) clearly holds — informally, T_{i+1} is "left-movable" into slot t. In the remainder of the proof, we assume that

$$r(T_{i+1}) > t. \tag{33}$$

Suppose, to the contrary of (P7), that there is no subtask V_k that is scheduled in S after slot t such that $r(V_k) \leq t$ (see Figure 12). By Claim 1, this implies that for each task V other than T and U, $lag(V, t) \leq -wt(V)$, where $wt(V)$ is the weight of V. (Note that, because $M = 2$, no such V has a subtask scheduled at time t.) Let

τ denote the set of all tasks. Then,

$$\sum\nolimits_{V \in \tau - \{T,U\}} lag(V,t) \leq -\sum\nolimits_{V \in \tau - \{T,U\}} wt(V). \tag{34}$$

By Claim 1 and our assumption that $\sum_{V \in \tau} wt(V) = M$, it follows that $\sum_{V \in \tau} lag(V,t) = 0$. Therefore,

$$\sum\nolimits_{V \in \tau - \{T,U\}} lag(V,t) = -lag(T,t) - lag(U,t) \tag{35}$$

Because $M = 2$, $\sum_{V \in \tau} wt(V) = 2$. This implies that

$$\sum\nolimits_{V \in \tau - \{T,U\}} wt(V) = 2 - wt(T) - wt(U) \tag{36}$$

By (34), (35), and (36), we have

$$lag(T,t) + lag(U,t) \geq 2 - wt(T) - wt(U). \tag{37}$$

Because $d(U_j) > t$, by Claim 1, we have $lag(U,t) < 1 - wt(U)$. Thus, by (37),

$$lag(T,t) > 1 - wt(T). \tag{38}$$

This implies that t is a pseudo-deadline of T, i.e., $d(T_i) = t$. We claim also that $b(T_i) = 1$. To see this, note that $b(T_i) = 0$ holds if and only if t is the last slot of a job of T. As explained in Section 2, each task has a lag of 0 at each slot that begins a new job. Thus, if $b(T_i) = 0$, then by Claim 1, $lag(T,t) = 1 - wt(T)$. This contradicts (38).

We have thus established that $d(T_i) = t$ and $b(T_i) = 1$. However, this implies that $r(T_{i+1}) = t$, which contradicts (33). Thus, our assumption that (P7) does not hold is false. This completes the proof. $\square$

(P8) Let S be a Pfair schedule for $M = 2$ processors. Consider time slots t and u, where $t < u$. Let A be the set of all subtasks scheduled by S in $[t, u]$. Suppose that there exists some task W that has no subtask in A. Then, there exists a subtask $T_i \in A$ such that $r(T_i) < t$ or $d(T_i) > u$.

Proof: Suppose, to the contrary, that

$$(\forall T_i : T_i \in A :: r(T_i) \geq t \ \wedge \ d(T_i) \leq u).$$

For any task V that has a subtask in A, let $V.n$ denote the number of such subtasks in A. Let V_k (respectively, V_l) be the first (respectively, last) subtask of V scheduled in the interval $[t, u]$. Then, $V.n = l - k + 1$. Because V is in A, $d(V_l) \leq u$ and $r(V_k) \geq t$. Thus, $d(V_l) - r(V_k) \leq u - t$. Therefore, by Equations (4) and (5) in Section

3, we have $\left(\left\lceil \frac{l \cdot V.p}{V.e} \right\rceil - 1\right) - \left\lfloor \frac{(k-1) \cdot V.p}{V.e} \right\rfloor \leq u - t$. Hence,

$$\left\lceil \frac{l \cdot V.p}{V.e} \right\rceil - \left\lfloor \frac{(k-1) \cdot V.p}{V.e} \right\rfloor \leq u - t + 1.$$

Because $\left\lceil \frac{l \cdot V.p}{V.e} \right\rceil \geq \frac{l \cdot V.p}{V.e}$ and $\left\lfloor \frac{(k-1) \cdot V.p}{V.e} \right\rfloor \leq \frac{(k-1) \cdot V.p}{V.e}$, we have

$$l \cdot \frac{V.p}{V.e} \; - \; (k-1) \cdot \frac{V.p}{V.e} \leq u - t + 1.$$

This implies that

$$\frac{V.p}{V.e} \leq \frac{u-t+1}{l-k+1},$$

which implies that

$$\frac{V.e}{V.p} \geq \frac{V.n}{u-t+1}.$$

From this, we conclude that

$$\sum_{V \text{ s.t. } V_i \in A} \frac{V.e}{V.p} \quad \geq \quad \sum_{V \text{ s.t. } V_i \in A} \frac{V.n}{u-t+1}.$$

Because there are a total of $2(u-t+1)$ slots in $[t, u]$ (recall that $M = 2$), we have

$$\sum_{V \text{ s.t. } V_i \in A} V.n = 2(u-t+1).$$

Therefore,

$$\sum_{V \text{ s.t. } V_i \in A} \frac{V.e}{V.p} \geq 2.$$

Given that there exists some task W that has no subtask in A, it follows that total utilization exceeds two. This contradicts the fact that the given task set is feasible for two processors. $\square$

Appendix B: PD Run-time Algorithm

In this appendix, we briefly describe the PD run-time algorithm proposed in [5]. This algorithm is shown in Figure 13.

The PD algorithm uses binomial heaps [7] to make scheduling decisions efficiently in $O(M \log N)$ time, where M is the number of processors in the system and N is the number of tasks. Multiple binomial heaps are used. These include **(i)** a heap H that stores the subtasks that are currently eligible to be scheduled, and **(ii)** a heap H_t for each future time slot t when one or more subtasks will become eligible. A subtask T_i is inserted into some H_t when its predecessor, T_{i-1}, is scheduled. Because there are N tasks, the number of non-empty binomial heaps is at most $N + 1$.

Binomial heaps require that a "key" be associated with each item to be stored. These keys determine the internal structure of the heap. In the PD algorithm, each subtask's "key" is its priority. Of course, PD priorities were assumed in [5], but the algorithm is still correct if the simpler priority definition given in Section 3 is used.

Binomial heaps support the following operations (in this list, H and H' denote arbitrary binomial heaps).

- *MakeHeap*(): Returns a pointer to an empty heap.
- *BuildHeap*(S): Takes a set of elements S as input and returns a heap containing those elements.
- *Insert*(H, T): Inserts task T into heap H.
- *ExtractMin*(H): Extracts the task with highest priority (minimum key) from heap H.
- *Union*(H, H'): Returns a heap that is the union of H and H'.

Makeheap takes $O(1)$ time and *BuildHeap* takes $O(N)$ time. The other operations take $O(\log N)$ time.

Each iteration of the main loop in lines (3)-(14) schedules M tasks in time slot t (for simplicity, it is assumed here that total utilization is M, so there are at least M eligible subtasks at all times). The processing of line (1), i.e., building an initial heap of N subtasks, takes $O(N)$ time. This is not included in the per-slot time complexity of the algorithm because this step is performed only once, at the start of the algorithm.

The per-slot time complexity is $O(M \log N)$ if we assume that the tests in lines (9) and (17) take $O(\log N)$ time. To see this, note that the **repeat** loop iterates M times at each step, and each iteration takes $O(\log N)$ time. To ensure that the tests at lines (9) and (17) take $O(\log N)$ time, an $O(\log N)$ search structure can be used that contains a pointer to each nonempty binomial heaps H_t, where $t \geq 1$. Examples of such structures include red-black trees and AVL trees.

```
Algorithm PD
begin
    H := BuildHeap(τ);
    t := 0;
    while true do
        repeat
            T := ExtractMin(H);
            "Schedule task T in slot t";
            t' := "the earliest future time at which task T will be eligible again";
            Requeue(T, t')
        until "M tasks have been scheduled in slot t";
        if "Heap H_{t+1} exists" then
            H := Union(H, H_{t+1})
        fi;
        t := t + 1
    od
end
```

```
Requeue(T, t)
begin
    if "Heap H_t does not exist" then
        H_t := MakeHeap()
    fi;
    "Determine T's priority at time t";
    Insert(H_t, T)
end
```

Figure 13: The PD run-time algorithm.

Appendix C: C Program for Generating Windows

```
/* This program generates the windows of a task with weight e/p, where *
 * e is the execution time requirement of the task and p is the period *
 * of the task.  This program can be downloaded from the web at the    *
 * following URL: http://www.cs.unc.edu/~anderson/ndraw.c              */

#include <stdio.h>
main(){
  int exec,period,gcd_of_ep;
  int i,j;
  int windowno;    /* represents the window numbers */
  int count;       /* represents the number of slots from the start of the current period */
  int start;       /* represents the number of slots before the start of the current period */
  int lag;         /* represents the (lag_value * period) */

  printf("\n--------------------------------------------------------------\n");
  printf("This program prints out the windows of a task in a period given its weight.\n");
  printf("The weight is assumed to be a fraction e/p, where e is the execution\n");
  printf("requirement and p is the period.\n");
  printf("\n--------------------------------------------------------------\n");

  while(1){
    printf("\nPlease enter the values of e and p (e = 0 to exit).\n");
    printf("e : ");
    scanf("%d",&exec);
    if (exec == 0)
      exit(0);
    printf("p : ");
    scanf("%d",&period);

    if (exec > period){
      printf("****************************************************\n");
      printf("The value of e should be smaller than the value of p\n");
      printf("****************************************************\n");
      continue;
    }
```

```
for (i=0; i<period; i++){
  printf("%2d ",i);
}
printf("\n------------------------------------------------------------\n");

/* Normalize the weight so that e and d are relatively prime. */
gcd_of_ep = gcd(period,exec);
exec = exec/gcd_of_ep;
period = period/gcd_of_ep;

/* Initialize values */
count = 0;
start = 0;
windowno = 1;
lag = 0;

/* Start printing out the windows, with the window of each subtask on a  *
 * different line.  The windows have to be repeated gcd number of times. */
for (i=1; i<=gcd_of_ep; i++){
  while(count < period){
    printf("%2d ",windowno);
    count++;
    lag += exec;
    if (lag >= period){
      /* a new window will start */
      printf("\n");
      for(j=1;j<start+count;j++)
        printf("   ");
      lag = lag - period - exec;
      windowno++;
      if (count != period)
        count--;
    }
  }
  /* Update values at the end of new period. */
```

```
      start += period;
      lag = 0;
      count = 0;
      printf("   ");
    }
  }
}

/* Calculates GCD of a and b. */
int gcd(int a, int b){

  if(b == 0)
    return a;
  else
    return gcd(b, a%b);
}
```