Proportionate Progress: A Notion of Fairness in Resource Allocation*

S. K. Baruah N. K. Cohen C. G. Plaxton D. A. Varvel

Department of Computer Science
University of Texas at Austin

September 5, 1994

Abstract

Given a set of n tasks and m resources, where each task x has a rational weight $x.w = x.e/x.p, 0 < x.w < 1$, a *periodic schedule* is one that allocates a resource to a task x for exactly $x.e$ time units in each interval $[x.p \cdot k, x.p \cdot (k+1))$ for all $k \in \mathbf{N}$. We define a notion of proportionate progress, called P-fairness, and use it to design an efficient algorithm which solves the periodic scheduling problem.

Keywords: Euclid's algorithm, fairness, network flow, periodic scheduling, resource allocation.

*This research was supported by NSF Research Initiation Award CCR–9111591, and Texas Advanced Research Program (TARP) Award #003658480. Contact author: Prof. Greg Plaxton, Department of Computer Science, University of Texas at Austin, Taylor Hall 2.124, Austin, Texas 78712–1188. Telephone: (512) 471–9751. FAX: (512) 471–8885. Email: plaxton@cs.utexas.edu.

1 Introduction

Scheduling is the act of assigning resources to activities or tasks. Scheduling problems typically involve a set of constraints (e.g., deadlines) that must be met by any schedule. Often these constraints are designed to enforce some notion of fairness; for example, a very weak fairness constraint might be that any task will eventually get to use the resource it has requested. For any particular set of constraints, there are two problems to be addressed: (i) the "decision" problem (i.e., determining whether or not a given instance is feasible), and (ii) the "scheduling" problem (i.e., actually constructing the schedule for a given feasible instance). Many sets of constraints result in an intractable decision problem [5].

The *periodic scheduling problem* was first discussed by Liu in 1969 [11]. Given a set of n tasks and m resources, where each task x has rational weight $x.w = x.e/x.p$, $0 < x.w < 1$, a *periodic schedule* is one that allocates a resource to a task x for exactly $x.e$ time units or *slots* in each interval $[x.p \cdot k, x.p \cdot (k+1))$ for all $k \in \mathbf{N}$. Scheduling decisions may be made only at integral times and a task may use either zero or one resources at a time.

We might also consider a relaxed version of the periodic scheduling problem in which tasks are not restricted to using zero or one resources at a time. Consider, for example, allowing resource sharing; that is, in each unit of time a task may use a fraction f of a resource, $0 \le f \le 1$. If $\sum_{x=0}^{n-1} x.w \le m$, the following straightforward "resource sharing" algorithm may be used to solve this relaxed version of the problem: Allocate a fraction $x.w$ of a resource to each task during each time unit. A second relaxed version requires integral resource usage, but allows a task to use more than one resource at a time (that is, to run with arbitrary concurrency). In this version, multiple-resource scheduling is easily reduced to single-resource scheduling.

There are several optimal single-resource scheduling algorithms for the periodic scheduling problem. The Earliest Deadline algorithm of Liu and Layland is one example [12]. None of them extends directly to multiple resources. As Liu pointed out, "the simple fact that a task can use only one [resource] even when several [resources] are free at the same time adds a surprising amount of difficulty" to the scheduling of multiple resources [11].

The decision problem has an efficient solution. Clearly systems in which $\sum_{x=0}^{n-1} x.w > m$ cannot be scheduled. If resource sharing is allowed, those in which $\sum_{x=0}^{n-1} x.w \le m$ can be scheduled by the resource sharing algorithm mentioned above. Baruah, Howell, and Rosier [1] used this fact, the network reduction of Horn [7], and the Ford-Fulkerson algorithm [4] to show that there are solutions to the periodic scheduling problem. Thus, the decision problem for such a periodic task system reduces to checking that $\sum_{x=0}^{n-1} x.w \le m$. A method similar to that of Baruah, Howell, and Rosier will be used in Section 3.

A more general form of the problem characterizes each task x by four parameters, $x.s$, $x.e$, $x.d$, and $x.p$, commonly referred to as starting time, execution requirement, deadline, and period, respectively. Here, a task x must receive exactly $x.e$ units of the resource in the time interval $[x.s + x.p \cdot k, x.s + x.p \cdot k + x.d)$ for all $k \in \mathbf{N}$. Leung's application of the Least Slack algorithm to this problem represents a recent improvement on Earliest Deadline [10], for the case where scheduling decisions are not required to occur at integer time instants. Leung was able to show that Least Slack schedules all instances that can be scheduled by Earliest Deadline, as well as some instances that Earliest Deadline cannot schedule. Both are optimal for scheduling a single resource but not for multiple resources; in fact, there is no known optimal algorithm for this problem. Our model may be viewed as a four-parameter model in which for all x, $x.s = 0$ and $x.d = x.p$. The general four-parameter model will not be addressed here.

Given a feasible instance of the periodic scheduling problem, a *schedule generation algorithm* performs a (possibly empty) *pre-processing phase* followed by an infinite *execution phase*. No output is produced during the pre-processing phase. During the execution phase, the algorithm produces an infinite sequence of outputs $\langle X_i : i \geq 0 \rangle$, where X_i is the subset of up to m tasks scheduled (i.e., assigned one copy of the resource) in slot i. (Note that any output-free prefix of the computation may be designated as the pre-processing phase.) Let t_i denote the elapsed running time (in the usual RAM model) between the beginning of the execution phase and the time at which output X_i is produced, $i \geq 0$. Also, let $t_{-1} = 0$. Then for any schedule generation algorithm $\mathcal{A}$, we define the *per-slot time complexity* of $\mathcal{A}$ as the maximum over all feasible instances and over all $i \geq 0$ of $t_i - t_{i-1}$. We further define the *pre-processing time complexity* of $\mathcal{A}$ as the maximum running time of the pre-processing phase over all feasible instances. A schedule generation algorithm is *polynomial time* if and only if both the per-slot time complexity and the pre-processing time complexity are polynomial in the input size. Prior to this paper, no polynomial-time schedule generation algorithm was known for the periodic scheduling problem.

Because all of the scheduling algorithms considered in this paper are schedule generation algorithms, for the sake of brevity we hereafter use the term "scheduling algorithm" to mean "schedule generation algorithm". Furthermore, because the scheduling algorithm that we consider has an empty pre-processing phase (i.e., pre-processing time complexity 0), we will focus our attention on slot time complexity. Accordingly, throughout the paper, every time bound given for a scheduling algorithm should be assumed to be a bound on the slot time complexity.

We solve the periodic scheduling problem by imposing an even stronger fairness constraint. Our approach is based on maintaining proportionate progress: each task is scheduled resources in proportion to its weight. Specifically, at every time t a task x must have been scheduled either $\lfloor x.w \cdot t \rfloor$ or $\lceil x.w \cdot t \rceil$ times. We call this *proportionate fairness* or *P-fairness*. P-fairness is a strictly stronger condition than periodic scheduling, in that any P-fair schedule is periodic while the converse is not generally true. P-fairness is a natural and desirable notion in certain practical applications. To the best of our knowledge, none of the scheduling algorithms currently known generate P-fair schedules even in the case of a single resource. We prove that any periodic scheduling problem instance for which $\sum_{x=0}^{n-1} x.w \leq m$ has a P-fair schedule. This proof makes use of certain results from network flow theory. We then describe and prove correct a polynomial-time scheduling algorithm that generates a P-fair schedule for any feasible instance. Since every P-fair schedule is also periodic, this algorithm solves the periodic scheduling problem.

We consider the research described here to be significant for several reasons. First, we introduce a new and potentially important notion of fairness in resource sharing, prove that this notion of fairness is actually achievable, and demonstrate its practical applicability. Second, as a corollary to our main results, we solve the periodic scheduling problem.

The remainder of this paper is organized as follows. Section 2 defines P-fairness and some related concepts and gives examples of practical applications of P-fairness. Section 3 establishes that P-fair schedules exist for the periodic scheduling problem. The algorithm corresponding to the proof has exponential time complexity, however. Section 4 proves the correctness of a simple algorithm for producing such P-fair schedules. The naive implementation of that algorithm schedules each slot in pseudo-polynomial time. Section 5 presents an example execution of our algorithm on a particular input instance. Section 6 proves the correctness of a polynomial-time implementation. Section 7 offers some concluding remarks.

2 P-Fairness

This section defines P-fairness and some related concepts. We start with some conventions:

- Scheduling decisions occur at integral values of *time*, numbered from 0. The real interval between time t and time $t+1$ (including t, excluding $t+1$) will be referred to as slot t, $t \in \mathbf{N}$.
- For integers a and b, let $[a, b) = \{a, \ldots, b-1\}$. Furthermore, let $[a, b] = [a, b+1)$, $(a, b] = [a+1, b+1)$, and $(a, b) = [a+1, b)$.
- We will consider an instance Φ of the fair resource sharing problem with m resources and n tasks. Specific tasks will be denoted by identifiers x and y, which range over Γ, the set of all tasks.
- Each task x has an integer *period* $x.p$, $x.p > 1$, an integer *execution requirement* $x.e$, $x.e \in (0, x.p)$, and a rational *weight* $x.w = x.e/x.p$. Note that $0 < x.w < 1$. Without loss of generality we confine our attention to the case where $\sum_{x \in \Gamma} x.w = m$.
- Let σ_i denote the ith symbol of string σ, $i \in \mathbf{N}$.

Now some definitions:

- A *schedule* S for instance Φ is a function from $\Gamma \times \mathbf{N}$ to $\{0, 1\}$, where $\sum_{x \in \Gamma} S(x, t) \leq m$, $t \in \mathbf{N}$. Informally, $S(x, t) = 1$ if and only if task x is scheduled in slot t.
- A schedule S is *periodic* if and only if

$$\forall i, x : i \in \mathbf{N}, x \in \Gamma : \textstyle\sum_{t \in [0, x.p \cdot i)} S(x, t) = x.e \cdot i.$$

- The *lag* of a task x at time t with respect to schedule S, denoted $\mathsf{lag}(S, x, t)$, is defined by:

$$\mathsf{lag}(S, x, t) = x.w \cdot t - \textstyle\sum_{i \in [0, t)} S(x, i).$$

- A schedule S is *P-fair* if and only if

$$\forall x, t : x \in \Gamma, t \in \mathbf{N} : -1 < \mathsf{lag}(S, x, t) < 1.$$

- A schedule S is *P-fair at time t* if and only if there exists a P-fair schedule S' such that

$$\forall x : x \in \Gamma : \mathsf{lag}(S, x, t) = \mathsf{lag}(S', x, t).$$

Informally, $\mathsf{lag}(S, x, t)$ measures the difference between the number of resource allocations that task x "should" have received in the set of slots $[0, t)$ and the number that it actually received.

Periodic schedules can also be defined in terms of lag constraints. In particular, a schedule S is periodic if and only if

$$\forall i, x : i \in \mathbf{N}, x \in \Gamma : \mathsf{lag}(S, x, x.p \cdot i) = 0,$$

from which it follows that *every P-fair schedule is periodic*. (Note that in the definition of lag, the term $x.w \cdot t$ is independent of S, and the term $\sum_{i \in [0,t)} S(x, i)$ is an integer.)

P-fairness is a very strict requirement. It demands that the absolute value of the difference between the expected allocation and the actual allocation to every task always be strictly less than 1. In other words, a task never gets an entire slot ahead or behind. In general it is not possible to guarantee a smaller variation in lag. Consider n identical tasks sharing a single resource, where the weight of each task is $1/n$. For n sufficiently large, we can make the lag of the first (resp., last) task scheduled come arbitrarily close to -1 (resp., 1).

P-fairness is the natural notion of fairness for many resource allocation problems. Here are two examples:

Example 1 An airline has m airplanes and n flight crews, $n > m$, all of which are based in the same city. Assume that exactly m flight crews are scheduled to work on any given day. Due to seniority, job performance, or other factors, it may be desirable to schedule some flight crews more often than others. For each flight crew x, set $x.w$ to the desired fraction of all days that x should work, while ensuring that $\sum_{x \in \Gamma} x.w = m$. A P-fair scheduler will produce a schedule in which every flight crew works at a steady rate: after t days, flight crew x will have worked either $\lfloor x.w \cdot t \rfloor$ or $\lceil x.w \cdot t \rceil$ days.

Example 2 Consider a node in a real-time communications network with a number of incoming and outgoing edges. The weight $x.w$ on an edge x corresponds to the relative amount of traffic expected on that edge. A P-fairness requirement may be necessary to maintain the real-time nature of the communications, and to prevent exceptionally long queueing delays from building up along certain edges.

To see how the concepts of lag and P-fairness arise within our (as yet undefined) scheduling algorithm for the periodic scheduling problem, the reader may now wish to briefly examine Section 5 before continuing with Section 3.

3 Existence of a P-Fair Schedule

In Sections 4 and 6 we will develop a polynomial-time P-fair scheduling algorithm. The proof of correctness of that algorithm relies on the existence of a P-fair schedule for the resource sharing problem. In this section we use a network flow argument to prove the existence of such a P-fair schedule. In principle, the network reduction could itself serve as the basis for a P-fair scheduling algorithm. Unfortunately, the size of the network generated by our reduction is exponential in the size of the given scheduling instance, and so the network reduction argument does not by itself provide a polynomial-time algorithm.

With respect to instance Φ of the resource scheduling problem, let $\mathsf{earliest}(x, j)$ (resp., $\mathsf{latest}(x, j)$) denote the earliest (resp., latest) slot during which task x may be scheduled for the jth time, $j \in \mathbf{N}$, in any P-fair schedule. We can easily derive closed form expressions for $\mathsf{earliest}(x, j)$ and $\mathsf{latest}(x, j)$. Note that $\mathsf{earliest}(x, j) = \min t : t \in \mathbf{N} : x.w \cdot (t+1) - (j+1) > -1$ and $\mathsf{latest}(x, j) = \max t : t \in \mathbf{N} : x.w \cdot t - j < 1$. Hence,

$$\begin{aligned} \mathsf{earliest}(x, j) &= \lfloor j/x.w \rfloor, \text{ and} \\ \mathsf{latest}(x, j) &= \lceil (j+1)/x.w \rceil - 1. \end{aligned}$$

Note that $\mathsf{earliest}(x, j) < \mathsf{latest}(x, j)$, $x \in \Gamma$, $j \in \mathbf{N}$. Furthermore, $\mathsf{earliest}(x, j+1) - \mathsf{latest}(x, j)$ is either 0 or 1. In other words, there is at most one slot where either the jth or the $j+1$st scheduling of task x may occur.

The remainder of this section is devoted to proving the existence of a P-fair schedule for any instance of the resource sharing problem Φ. Our proof strategy is as follows: First, we will describe a reduction from instance Φ to a weighted digraph G with a designated source and sink, such that certain flows in G correspond exactly (in a manner that will be made precise) to a P-fair schedule for Φ. Then we will prove the existence of such a flow in G.

Throughout this section, let L denote the least common multiple of the task periods: $L = \mathrm{lcm}_{x \in \Gamma}\, x.p$.

Lemma 3.1 Instance Φ has a P-fair schedule if and only if there exists a schedule S such that

$$\forall x, t : x \in \Gamma, t \in (0, L] : -1 < \mathsf{lag}(S, x, t) < 1.$$

Proof: An infinite P-fair schedule S' may be obtained from S by scheduling in slot t those tasks scheduled by S in slot $t \bmod L$. $\square$

Theorem 1 Instance Φ has a P-fair schedule.

Before proving this theorem, we will present some definitions and an important lemma.

Recall that $x.w = x.e/x.p$. We describe below the construction of a weighted digraph G. The vertex set V of G is the union of 6 disjoint sets of vertices $V_0, \ldots, V_5$, and the edge set E of G is the union of 5 disjoint sets of edges $E_0, \ldots, E_4$, where E_i is a subset of $(V_i \times V_{i+1} \times \mathbf{N})$, $0 \le i \le 4$. That is, G is a "6-layered" graph, with all edges connecting vertices in adjacent layers. The sets of vertices are as follows:

$$\begin{aligned} V_0 &= \{\mathsf{source}\}, \\ V_1 &= \{\langle 1, x \rangle \mid x \in \Gamma)\}, \\ V_2 &= \{\langle 2, x, j \rangle \mid x \in \Gamma, j \in [0, x.w \cdot L)\}, \\ V_3 &= \{\langle 3, x, t \rangle \mid x \in \Gamma, t \in [0, L)\}, \\ V_4 &= \{\langle 4, t \rangle \mid t \in [0, L)\}, \text{ and} \\ V_5 &= \{\mathsf{sink}\}. \end{aligned}$$

An edge is represented by a 3-tuple. For $u, v \in V$ and $w \in \mathbf{N}$, the 3-tuple $(u, v, w) \in E$ represents an edge from u to v of capacity w. The sets of edges in G are as follows:

$$\begin{aligned}
E_0 &= \{(\mathsf{source}, \langle 1, x\rangle, x.w \cdot L) \mid x \in \Gamma\}, \\
E_1 &= \{(\langle 1, x\rangle, \langle 2, x, j\rangle, 1) \mid x \in \Gamma, j \in [0, x.w \cdot L)\}, \\
E_2 &= \{(\langle 2, x, j\rangle, \langle 3, x, t\rangle, 1) \mid x \in \Gamma, \\
&\qquad j \in [0, x.w \cdot L), t \in [\mathsf{earliest}(x, j), \mathsf{latest}(x, j)]\}, \\
E_3 &= \{(\langle 3, x, t\rangle, \langle 4, t\rangle, 1) \mid x \in \Gamma, t \in [0, L)\}, \text{ and} \\
E_4 &= \{(\langle 4, t\rangle, \mathsf{sink}, m) \mid t \in [0, L)\}.
\end{aligned}$$

Lemma 3.2 If there is an integral flow of size $m \cdot L$ in G, then there exists a P-fair schedule for Φ.

Proof: By Lemma 3.1, it suffices to prove that the existence of an integral flow of size $m \cdot L$ in G implies the existence of a schedule S for Φ such that

$$\forall x, t : x \in \Gamma, t \in (0, L] : -1 < \mathsf{lag}(S, x, t) < 1.$$

Suppose there is an integral flow of size $m \cdot L$ in G. The total capacity of E_0, the set of edges leading out of the source vertex, is equal to $\sum_{x \in \Gamma} x.w \cdot L = m \cdot L$. Hence, each edge in E_0 is filled to capacity, and each vertex $\langle 1, x\rangle$ receives exactly $x.w \cdot L$ units of flow. Since there are $x.w \cdot L$ vertices in V_2 each connected (by an edge of unit capacity) to vertex $\langle 1, x\rangle$, and no two vertices in V_1 are connected to the same vertex in V_2, it follows that each vertex in V_2 receives a unit flow. Accordingly, each vertex in V_2 sends a unit flow to some vertex in V_3.

We will construct the desired schedule S from the given flow according to the following rule: Allocate a resource to task x in slot t if and only if there is a unit flow from vertex $\langle 2, x, j\rangle$ to vertex $\langle 3, x, t\rangle$.

Because the total flow into the sink vertex is $m \cdot L$, each of the L edges of capacity m in E_4 carries m units of flow. Hence, for all $t \in [0, L)$, vertex $\langle 4, t\rangle$ receives exactly m unit flows from vertices in V_3. Each vertex $\langle 3, x, t\rangle$ in V_3 is connected (by an edge of unit capacity) to vertex $\langle 4, t\rangle$, and is not connected to any other vertex in V_4. Thus, S schedules exactly m tasks in each time slot t, for all $t \in [0, L)$. To see that no lag constraints are violated by S, observe that for each task x and for all $j \in [0, x.w \cdot L)$, the jth scheduling of task x occurs at a slot in the interval $[\mathsf{earliest}(x, j), \mathsf{latest}(x, j)]$. (The jth scheduling corresponds to the unique unit flow out of vertex $\langle 2, x, j\rangle$.) □

We will now show the existence of an integral flow.

Proof of Theorem 1: Since all edges of the graph have integral capacity, if there is a fractional flow of size $m \cdot L$ in the graph then there is an integral flow of that size [4]. It remains to be shown that such a fractional flow exists. We use the following flow assignments:

- Each edge $(\mathsf{source}, \langle 1, x\rangle, x.w \cdot L) \in E_0$ carries a flow of $x.w \cdot L$.
- Each edge $(\langle 1, x\rangle, \langle 2, x, j\rangle, 1) \in E_1$ carries a unit flow.
- Each edge $(\langle 3, x, t\rangle, \langle 4, t\rangle, 1) \in E_3$ carries a flow of size $x.w$.

- Each edge $(\langle 4,t\rangle, \mathsf{sink}, m) \in E_4$ carries a flow of size m.
- The flows through edges in E_2 are as follows:
  - Each edge $(\langle 2,x,j\rangle, \langle 3,x,\mathsf{earliest}(x,j)\rangle, 1)$ carries a flow of size
  $$x.w - (j - x.w \cdot \lfloor j/x.w \rfloor),$$
  which is less than 1, the capacity of the edge.
  - Each edge $(\langle 2,x,j\rangle, \langle 3,x,\mathsf{latest}(x,j)\rangle, 1)$ such that $\mathsf{latest}(x,j) = \mathsf{earliest}(x,j+1)$ carries a flow of size
  $$(j+1) - x.w \cdot \lfloor (j+1)/x.w \rfloor,$$
  which is also less than 1, the capacity of the edge.
  - Every other edge $(\langle 2,x,j\rangle, \langle 3,x,t\rangle, 1) \in E_2$ carries a flow of size $x.w$.

We will now prove that the flow just defined is a valid flow of size $m \cdot L$. The capacity constraints have been met. The flow out of the source vertex is $\sum_{x\in\Gamma}(x.w \cdot L) = m \cdot L$. We will now complete the proof by showing that flow is conserved at every interior vertex.

The flow into each vertex in V_1 is $x.w \cdot L$, and there are $x.w \cdot L$ edges leaving, each carrying a unit flow. The flow into each vertex in V_2 is 1. Below we will prove that the flow out of each vertex in V_2 is 1, and that the flow into each vertex in V_3 is $x.w$. Each vertex in V_3 has only one outgoing edge carrying a flow of $x.w$. Each vertex in V_4 has n incoming edges each carrying a flow of size $x.w$; since $\sum_{x\in\Gamma} x.w = m$, the flow in is m, which equals the flow out on the one outgoing edge.

It remains to prove that: (i) the flow out of each vertex in V_2 is 1, and (ii) the flow into each vertex in V_3 is $x.w$.

For (i), consider an arbitrary vertex $\langle 2,x,j\rangle$ in V_2. There are $\mathsf{latest}(x,j) - \mathsf{earliest}(x,j) + 1$, or $\lceil (j+1)/x.w \rceil - \lfloor j/x.w \rfloor$, outgoing edges from $\langle 2,x,j\rangle$. If $\mathsf{earliest}(x,j+1) = \mathsf{latest}(x,j)$ (equivalently, $\lceil (j+1)/x.w \rceil - 1 = \lfloor (j+1)/x.w \rfloor$), then the flow out of $\langle 2,x,j\rangle$ is

$$\begin{aligned} &x.w - (j - x.w \cdot \lfloor j/x.w \rfloor) + x.w \cdot (\lceil (j+1)/x.w \rceil - \lfloor j/x.w \rfloor - 2) \\ &+ (j+1) - x.w \cdot \lfloor (j+1)/x.w \rfloor, \end{aligned}$$

which simplifies to 1. Otherwise, $\mathsf{earliest}(x,j+1) = \mathsf{latest}(x,j) + 1$ (equivalently, $\lceil (j+1)/x.w \rceil = \lfloor (j+1)/x.w \rfloor = (j+1)/x.w$), and the flow out of $\langle 2,x,j\rangle$ is

$$x.w - (j - x.w \cdot \lfloor j/x.w \rfloor) + x.w \cdot (\lceil (j+1)/x.w \rceil - \lfloor j/x.w \rfloor - 1),$$

which also simplifies to 1.

For (ii), consider an arbitrary vertex $\langle 3,x,t\rangle$ in V_3. If $t = \mathsf{latest}(x,j) = \mathsf{earliest}(x,j+1)$ for some $j \in \mathbf{N}$, then there are two incoming edges to $\langle 3,x,t\rangle$, namely $(\langle 2,x,j\rangle, \langle 3,x,t\rangle, 1)$ and $(\langle 2,x,j+1\rangle, \langle 3,x,t\rangle, 1)$. These edges carry flows of size $(j+1) - x.w \cdot \lfloor (j+1)/x.w \rfloor$ and $x.w - ((j+1) - x.w \cdot \lfloor (j+1)/x.w \rfloor)$, respectively, for a total incoming flow of $x.w$. Otherwise, there is only one incoming edge to $\langle 3,x,t\rangle$, and it carries a flow of $x.w$. $\square$

4 A P-Fair Scheduling Algorithm

In this section, we present our scheduling algorithm, Algorithm PF, and prove that it produces a P-fair schedule. Algorithm PF has the following high-level structure: At each time $t \geq 0$, a dynamic priority is assigned to each task and the m highest-priority tasks are scheduled in slot t (ties are broken arbitrarily). An example execution of our scheduling algorithm is presented in Section 5. The reader may wish to briefly examine that example before continuing with the formal presentation that follows.

First, some definitions:

- The *characteristic string* of task x, denoted $\alpha(x)$, is an infinite string over $\{-, 0, +\}$ with
$$\alpha_t(x) = \text{sign}(x.w \cdot (t+1) - \lfloor x.w \cdot t \rfloor - 1), t \in \mathbf{N}.$$
- The *characteristic substring* of task x at time t is the finite string
$$\alpha(x,t) \stackrel{\text{def}}{=} \alpha_{t+1}(x)\alpha_{t+2}(x) \cdots \alpha_{t'}(x),$$
where $t' = \min i : i > t : \alpha_i(x) = 0$.
- With respect to P-fair schedule S at time t, we say that: task x is *ahead* if and only if $\mathsf{lag}(S,x,t) < 0$; task x is *behind* if and only if $\mathsf{lag}(S,x,t) > 0$; task x is *punctual* if and only if it is neither ahead nor behind.
- With respect to P-fair schedule S at time t, we say that: task x is *tnegru* if and only if x is ahead and $\alpha_t(x) \neq +$; task x is *urgent* if and only if x is behind and $\alpha_t(x) \neq -$; task x is *contending* if and only if it is neither tnegru nor urgent.

Lemmas 4.1 to 4.5 provide the logical machinery that we will need in order to reason about the terms introduced above.

Lemma 4.1 If task x is ahead at time t under P-fair schedule S, then:

(a) If $\alpha_t(x) = -$, then $S(x,t) = 0$ and task x is ahead at time $t+1$.

(b) If $\alpha_t(x) = 0$, then $S(x,t) = 0$ and task x is punctual at time $t+1$.

(c) If $\alpha_t(x) = +$ and $S(x,t) = 1$, then task x is ahead at time $t+1$.

(d) If $\alpha_t(x) = +$ and $S(x,t) = 0$, then task x is behind at time $t+1$.

Proof: Assuming that task x is ahead at time t under P-fair schedule S, we have $\sum_{i \in [0,t)} S(x,i) = \lceil x.w \cdot t \rceil$, where $x.w \cdot t \notin \mathbf{N}$; hence, $\lceil x.w \cdot t \rceil = \lfloor x.w \cdot t \rfloor + 1$ and $\lfloor x.w \cdot t \rfloor = \sum_{i \in [0,t)} S(x,i) - 1$.

We now deal with each part in turn. For Part (a) we have

$$\begin{aligned}
& \alpha_t(x) = - \quad \wedge \quad \textstyle\sum_{i \in [0,t)} S(x,i) = \lceil x.w \cdot t \rceil \\
\Longrightarrow \quad & x.w \cdot (t+1) - \lfloor x.w \cdot t \rfloor - 1 < 0 \quad \wedge \quad \textstyle\sum_{i \in [0,t)} S(x,i) = \lceil x.w \cdot t \rceil \\
\Longrightarrow \quad & x.w \cdot (t+1) - \textstyle\sum_{i \in [0,t)} S(x,i) < 0 \\
\Longrightarrow \quad & \mathsf{lag}(S,x,t+1) + S(x,t) < 0.
\end{aligned}$$

Because schedule S is P-fair, $\mathsf{lag}(S,x,t+1) > -1$, and the inequality $\mathsf{lag}(S,x,t+1)+S(x,t) < 0$ implies that $S(x,t)=0$. Hence, $\mathsf{lag}(S,x,t+1) < 0$ and task x is ahead at time $t+1$, as required. For Part (b) we have

$$\begin{array}{rl} & \alpha_t(x)=0 \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lceil x.w\cdot t\rceil \\ \Longrightarrow & x.w\cdot(t+1) - \lfloor x.w\cdot t\rfloor - 1 = 0 \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lceil x.w\cdot t\rceil \\ \Longrightarrow & x.w\cdot(t+1) - \sum_{i\in[0,t)} S(x,i) = 0 \\ \Longrightarrow & \mathsf{lag}(S,x,t+1) + S(x,t) = 0. \end{array}$$

Note that $\mathsf{lag}(S,x,t+1)+S(x,t)=0$ implies that $S(x,t)=0$ and task x is punctual at time $t+1$, as required. For Part (c), note that if $S(x,t)=1$ then $\mathsf{lag}(S,x,t+1) < \mathsf{lag}(S,x,t)$. Finally, for Part (d) we have

$$\begin{array}{rl} & \alpha_t(x)=+ \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lceil x.w\cdot t\rceil \quad \wedge \quad S(x,t)=0 \\ \Longrightarrow & x.w\cdot(t+1) - \lfloor x.w\cdot t\rfloor - 1 > 0 \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lceil x.w\cdot t\rceil \quad \wedge \quad S(x,t)=0 \\ \Longrightarrow & x.w\cdot(t+1) - \sum_{i\in[0,t)} S(x,i) > 0 \quad \wedge \quad S(x,t)=0 \\ \Longrightarrow & \mathsf{lag}(S,x,t+1) > 0. \end{array}$$

□

Lemma 4.2 If task x is behind at time t under P-fair schedule S, then:

(a) If $\alpha_t(x) = -$ and $S(x,t)=1$, then task x is ahead at time $t+1$.

(b) If $\alpha_t(x) = -$ and $S(x,t)=0$, then task x is behind at time $t+1$.

(c) If $\alpha_t(x) = 0$, then $S(x,t)=1$ and task x is punctual at time $t+1$.

(d) If $\alpha_t(x) = +$, then $S(x,t)=1$ and task x is behind at time $t+1$.

Proof: Assuming that task x is behind at time t under P-fair schedule S, we have $\sum_{i\in[t]} S(x,i) = \lfloor x.w\cdot t\rfloor$, where $x.w\cdot t \notin \mathbf{N}$. Again, we deal with each part in turn. For Part (a) we have

$$\begin{array}{rl} & \alpha_t(x)=- \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lfloor x.w\cdot t\rfloor \quad \wedge \quad S(x,t)=1 \\ \Longrightarrow & x.w\cdot(t+1) - \lfloor x.w\cdot t\rfloor - 1 < 0 \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lfloor x.w\cdot t\rfloor \quad \wedge \quad S(x,t)=1 \\ \Longrightarrow & x.w\cdot(t+1) - \sum_{i\in[0,t)} S(x,i) - 1 < 0 \quad \wedge \quad S(x,t)=1 \\ \Longrightarrow & \mathsf{lag}(S,x,t+1) < 0. \end{array}$$

For Part (b), note that if $S(x,t)=0$ then $\mathsf{lag}(S,x,t+1) > \mathsf{lag}(S,x,t)$. For Part (c) we have

$$\begin{array}{rl} & \alpha_t(x)=0 \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lfloor x.w\cdot t\rfloor \\ \Longrightarrow & x.w\cdot(t+1) - \lfloor x.w\cdot t\rfloor - 1 = 0 \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lfloor x.w\cdot t\rfloor \\ \Longrightarrow & x.w\cdot(t+1) - \sum_{i\in[0,t)} S(x,i) - 1 = 0 \\ \Longrightarrow & \mathsf{lag}(S,x,t+1) + S(x,t) - 1 = 0. \end{array}$$

Note that $\mathsf{lag}(S, x, t+1) + S(x,t) - 1 = 0$ implies $S(x,t) = 1$ and task x is punctual at time $t+1$, as required. For Part (d) we have

$$\begin{array}{rl} & \alpha_t(x) = + \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lfloor x.w \cdot t \rfloor \\ \Longrightarrow & x.w \cdot (t+1) - \lfloor x.w \cdot t \rfloor - 1 > 0 \quad \wedge \quad \sum_{i\in[0,t)} S(x,i) = \lfloor x.w \cdot t \rfloor \\ \Longrightarrow & x.w \cdot (t+1) - \sum_{i\in[0,t)} S(x,i) - 1 > 0 \\ \Longrightarrow & \mathsf{lag}(S, x, t+1) + S(x,t) - 1 > 0. \end{array}$$

Note that $\mathsf{lag}(S, x, t+1) + S(x,t) - 1 > 0$ implies $S(x,t) = 1$ and task x is behind at time $t+1$, as required. □

Lemma 4.3 If task x is tnegru at time t under P-fair schedule S, then $S(x,t) = 0$.

Proof: Follows from Lemma 4.1(a) and (b). □

Lemma 4.4 If task x is urgent at time t under P-fair schedule S, then $S(x,t) = 1$.

Proof: Follows from Lemma 4.2(c) and (d). □

Lemma 4.5 If task x is contending at time t under P-fair schedule S, then:

(a) If $S(x,t) = 1$, then x is ahead at time $t+1$.

(b) If $S(x,t) = 0$, then x is behind at time $t+1$.

Proof: If x is ahead at time t, this follows from Lemma 4.1(c) and (d) and if x is behind at time t it follows from Lemma 4.2(a) and (b). For x punctual we have the following:

$$\begin{array}{rl} & \mathsf{lag}(S, x, t) = x.w \cdot t - \sum_{i\in[0,t)} S(x,i) = 0 \\ \Longrightarrow & x.w \cdot (t+1) - \sum_{i\in[0,t)} S(x,i) = x.w \\ \Longrightarrow & \mathsf{lag}(S, x, t+1) = x.w - S(x,t). \end{array}$$

Because $0 < x.w < 1$, if $S(x,t) = 1$ then x is ahead at time $t+1$, and if $S(x,t) = 0$ then x is behind at time $t+1$, as required. □

Given the preceding definitions and lemmas, it is now straightforward to present our scheduling algorithm, which will be referred to as *Algorithm PF*. At any time t the task of Algorithm PF is to determine which m-subset of the n tasks to schedule. By Lemma 4.4, every urgent task must be scheduled in the current time slot in order to preserve P-fairness. Symmetrically, Lemma 4.3 implies that no tnegru task can be scheduled in the current time slot without violating P-fairness. Since our goal is to prove that Algorithm PF produces a P-fair schedule, it must be that Algorithm PF schedules all of the urgent tasks and none of the tnegru tasks. It remains to define the behavior of Algorithm PF on the set of contending tasks.

Before doing so, however, we should pause to address two possible pitfalls. Let n_0, n_1, and n_2 denote the number of tnegru, contending, and urgent tasks at time t, respectively. If

$n_2 > m$, then it would be impossible for Algorithm PF to schedule all of the urgent tasks. Symmetrically, if $n_0 > n - m$, then Algorithm PF would be forced to schedule some tnegru task. (Because $\sum_{x \in [0,n)} x.w = m$, we cannot hope to schedule instance Φ correctly unless all m resources are allocated in every slot.) An immediate consequence of Theorem 2, stated below, is that neither of these pitfalls will ever arise under Algorithm PF. Thus, in defining the behavior of Algorithm PF on the set of contending tasks, we can assume that $n_0 \le n - m$ and $n_2 \le m$. The task of Algorithm PF is to determine which subset (of size $m - n_2 \le n_1$) of the n_1 contending tasks to schedule.

At each time t, we can define a total order $\succeq$ on the set of contending tasks as follows: $x \succeq y$ if and only if $\alpha(x,t) \ge \alpha(y,t)$, where the comparison between characteristic substrings $\alpha(x,t)$ and $\alpha(y,t)$ is resolved lexicographically with $- < 0 < +$. Ties can be broken arbitrarily; for example, we could assume that ties are broken in favor of the lower-numbered task.

Algorithm PF schedules the $m - n_2$ highest-priority contending tasks according to this total order. Algorithm PF is summarized in its entirety below:

1. Schedule all urgent tasks.
2. Allocate the remaining resources to the highest-priority contending tasks according to the total order $\succeq$.

Throughout the remainder of this section, let S_{PF} denote the schedule produced by Algorithm PF on instance Φ.

Lemma 4.6 If schedule S_{PF} is P-fair at time t, then it is P-fair at time $t+1$, $t \in \mathbf{N}$.

Proof: Assume that schedule S_{PF} is P-fair at time t for some $t \in \mathbf{N}$. Hence, there exists a P-fair schedule S such that $\mathsf{lag}(S_{PF}, x, t) = \mathsf{lag}(S, x, t)$, $x \in [0, n)$. Let X (resp., Y) denote the m-subset of tasks scheduled by S (resp., S_{PF}) in slot t. If $X = Y$ then S_{PF} is P-fair at time $t+1$ because S is P-fair at time $t+1$. If $X \ne Y$, there exist tasks $x \in X$ and $y \in Y$ such that $x \in X \setminus Y$ and $y \in Y \setminus X$. In the argument that follows we will demonstrate the existence of a P-fair schedule S' such that: (i) $\mathsf{lag}(S_{PF}, x, t) = \mathsf{lag}(S', x, t)$, $x \in [0, n)$, and (ii) S' schedules the m-subset $X \setminus \{x\} \cup \{y\}$ in slot t. By repeating this argument $|X \setminus Y|$ times, we can obtain a sequence of P-fair schedules such that the last P-fair schedule in the sequence, S^*, satisfies $\mathsf{lag}(S_{PF}, x, t+1) = \mathsf{lag}(S^*, x, t+1)$. Hence, schedule S_{PF} is P-fair at time $t+1$, proving the lemma.

Accordingly, it is sufficent to prove existence of a P-fair schedule S' as defined above. We begin by claiming that $S(x,i) \ne S(y,i)$ for some $i > t$. (If not, it follows easily that $x.w = y.w$, $\mathsf{lag}(S, x, t) = \mathsf{lag}(S, y, t) + 1$, and hence that Algorithm PF would have given priority to task x over task y at time t, a contradiction.) We transform schedule S into S' as follows. Let

$$t' \stackrel{\text{def}}{=} \min i : i > t : S(x,i) \ne S(y,i).$$

We prove below that in fact $S(x,t') = 0$ and $S(y,t') = 1$. Schedule S' is defined to be identical to S except that we "swap" the allocations to tasks x and y at slots t and t', setting $S'(x,t) = 0$, $S'(y,t) = 1$, $S'(x,t') = 1$, and $S'(y,t') = 0$.

In the arguments that follow, all statements "categorizing" tasks x and y (e.g., "task x is not urgent at time t") are being made with respect to the P-fair schedule S. (Note that at time t, it makes no difference whether our claims are made with respect to S_{PF} or S, since $\mathsf{lag}(S_{PF}, x, t) = \mathsf{lag}(S, x, t)$, $x \in [0, n)$.)

Consider the following predicates:

$$
\begin{aligned}
P_0(i) &\stackrel{\text{def}}{=} \text{task } x \text{ is ahead at time } i, \\
P_1(i) &\stackrel{\text{def}}{=} \text{task } y \text{ is behind at time } i, \\
P_2(i) &\stackrel{\text{def}}{=} \alpha_j(x) = \alpha_j(y) \neq 0,\ \ j \in (t, i), \text{ and} \\
P(i) &\stackrel{\text{def}}{=} P_0(i) \wedge P_1(i) \wedge P_2(i).
\end{aligned}
$$

We will prove by induction on i that $P(i)$ holds, $i \in (t, t']$. For the base case, set $i = t+1$. Since $S_{PF}(x, t) = 0$ (resp., $S(y, t) = 0$), task x (resp., y) is not urgent at time t. Similarly, since $S(x, t) = 1$ (resp., $S_{PF}(y, t) = 1$), task x (resp., y) is not tnegru at time t. Hence, tasks x and y are both contending at time t. We can now use Lemma 4.5(a) to establish $P_0(t+1)$. Similarly, Lemma 4.5(b) implies $P_1(t+1)$. Note that $P_2(t+1)$ is vacuously true. This completes the base case of the induction.

For the induction step, we assume that $P(i)$ holds over the interval $(t, i]$, and prove that it holds over $(t, i+1]$, where $i \in (t, t')$. By the definition of t', we have

$$S(x, i) = S(y, i). \tag{1}$$

Given that $P_2(i)$ is part of our induction hypothesis, $P_2(i+1)$ will follow if we can establish that

$$\alpha_i(x) = \alpha_i(y) \neq 0. \tag{2}$$

Assuming that Equation (2) fails to hold, there are four cases to consider: (i) $\alpha_i(x) = -$ and $\alpha_i(y) = 0$, (ii) $\alpha_i(x) = -$ and $\alpha_i(y) = +$, (iii) $\alpha_i(x) = 0$ and $\alpha_i(y) = +$, and (iv) $\alpha_i(x) = 0$ and $\alpha_i(y) = 0$. (The symmetric versions of the first three cases are impossible since $y \succeq x$ at time t and $P_2(i)$ holds.) Assume that Case (i) holds. Lemma 4.1(a) and $P_0(i)$ imply that $S(x, i) = 0$. Lemma 4.2(c) and $P_1(i)$ imply that $S(y, i) = 1$, contradicting Equation (1). Assume that Case (ii) holds. Lemma 4.1(a) and $P_0(i)$ imply that $S(x, i) = 0$. Lemma 4.2(d) and $P_1(i)$ imply that $S(y, i) = 1$, contradicting Equation (1). Assume that Case (iii) holds. Lemma 4.1(b) and $P_0(i)$ imply that $S(x, i) = 0$. Lemma 4.2(d) and $P_1(i)$ imply that $S(y, i) = 1$, contradicting Equation (1). Finally, assume that Case (iv) holds. Lemma 4.1(b) and $P_0(i)$ imply that $S(x, i) = 0$. Lemma 4.2(c) and $P_1(i)$ imply that $S(y, i) = 1$, contradicting Equation (1). Hence, Equation (2) holds and $P_2(i+1)$ holds. It remains to establish $P_0(i+1)$ and $P_1(i+1)$.

By Lemma 4.1 and $P_0(i)$, if $S(x, i) = 1$ then $\alpha_i(x) = +$. Conversely, $\alpha_i(x) = +$ and Equation (2) imply $\alpha_i(y) = +$; $P_1(i)$ and Lemma 4.2(d) then imply $S(y, i) = 1$; and finally, Equation (1) implies $S(x, i) = 1$. Hence, we have proven that

$$S(x, i) = 1 \iff \alpha_i(x) = +. \tag{3}$$

By Equations (1), (2), and (3), at time i we either had: (i) $S(x, i) = S(y, i) = 0$ and $\alpha_i(x) = \alpha_i(y) = -$, or (ii) $S(x, i) = S(y, i) = 1$ and $\alpha_i(x) = \alpha_i(y) = +$. Consider Case (i).

By Lemma 4.1(a) and $P_0(i)$, $P_0(i+1)$ holds. By Lemma 4.2(b) and $P_1(i)$, $P_1(i+1)$ holds. Similarly, consider Case (ii). By Lemma 4.1(c) and $P_0(i), P_0(i+1)$ holds. By Lemma 4.2(d) and $P_1(i), P_1(i+1)$ holds. Hence, $P_0(i+1)$ and $P_1(i+1)$ hold. This completes our proof by induction.

Given that $P(i)$ holds, $i \in (t, t']$, it is now quite easy to prove the two remaining claims that we need, namely: (i) $S(x, t') = 0$ and $S(y, t') = 1$, and (ii) S' is P-fair. Because our algorithm schedules task y,

$$\alpha_{t'}(x) \leq \alpha_{t'}(y), \tag{4}$$

and by the definition of t',

$$S(x, t') \neq S(y, t'). \tag{5}$$

If $\alpha_{t'}(x) = -$ or $\alpha_{t'}(x) = 0$, then $P_0(t')$ and Lemma 4.1 imply $S(x, t') = 0$, and so Equation (5) implies $S(y, t') = 1$. If $\alpha_{t'}(x) = +$, then Equation (4) implies $\alpha_{t'}(y) = +$; Lemma 4.2(d) then implies $S(y, t') = 1$; and finally, Equation (5) implies $S(x, t') = 0$. Thus Claim (i) holds.

For Claim (ii), it is sufficient to prove that

$$\forall i : i \in (t, t'] : -1 < \mathsf{lag}(S', x, i) < 1 \tag{6}$$

and

$$\forall i : i \in (t, t'] : -1 < \mathsf{lag}(S', y, i) < 1, \tag{7}$$

since all other lags are the same as under schedule S. Note that $\mathsf{lag}(S', x, i) = \mathsf{lag}(S, x, i) + 1$, $i \in (t, t']$. Since S is P-fair, Equation (6) will hold if we can show that $\mathsf{lag}(S, x, i) < 0$, $i \in (t, t']$. This is immediate, since $P_0(i)$ holds for all $i \in (t, t']$. Thus, Equation (6) holds. A symmetric argument proves that Equation (7) holds. Hence, Claim (ii) holds, and our proof is complete. □

Theorem 2 Schedule S_{PF} is P-fair.

Proof: By Theorem 1, schedule S_{PF} is P-fair at time 0. Hence, Lemma 4.6 implies that schedule S_{PF} is P-fair at time t, $t \in \mathbf{N}$. □

5 An Example Execution of Algorithm PF

In this section we trace in considerable detail the execution of Algorithm PF on a particular input instance. The input instance is presented in Table 1. There are four tasks: v, w, x, and y. The tasks are to be scheduled on three resources. Values of the relevant task parameters are given in Table 1. Each task is characterized by an execution requirement (see the second column of Table 1) and a period (see the third column). The weight of a task is defined to be the ratio of its execution requirement to its period (see the fourth column). Since the weights of the four tasks sum to less than the number of copies of the resource, a fifth "dummy" task z has been added with weight such that the weights of all tasks, including the dummy, together sum to 3.

A couple of remarks concerning the input instance are in order. First, we have represented the weights of the tasks in Table 1 in decimal as well as fractional form; since the tasks x,

task	execution requirement	period	weight
v	1	3	$1/3 = 0.\overline{3}$
w	2	4	$2/4 = 0.5$
x	5	7	$5/7 = 0.\overline{714285}$
y	8	11	$8/11 = 0.\overline{72}$
z	335	462	$335/462 = 0.7\overline{251082}$

Table 1: An example instance of the periodic scheduling problem.

y, and z have weights that are very close to each other, the decimal expansion is useful to make the relative ordering of these weights more obvious. Second, observe that the dummy task z has a very large period relative to those of the other tasks. (In general, the period of the dummy task can be as large as the least common multiple of the other task periods.) We would like to emphasize that the inclusion of a dummy task does not degrade the per-slot time complexity of Algorithm PF by more than a constant factor. (Algorithm PF requires time linear in the size of the input instance to schedule each slot, and the inclusion of task z increases the input size by no more than a constant factor.)

In Table 2, we trace the execution of Algorithm PF on the above input instance for the first twenty time units.

- The first column of the table indicates the time associated with each row.
- In the next set of five columns we list, for each of the five tasks, the product of the task's lag and its period at each time slot. Since all lags are initially zero, and the lag of a given task u changes by either subtracting $(1 - u.w)$ (when u is allocated a resource), or by adding $u.w$ (when u is not allocated a resource), representing the lag in this manner (i.e., as a product of lag and period) ensures that all entries are integers.
- In the next set of five columns we list the first twenty symbols in the characteristic string of each task. The characteristic string of any task may be obtained from the table by reading down the appropriate column. (For example, the characteristic string of task v is of the form "$- - 0 - - 0 \ldots$", and that of the dummy task z begins "$- + + - + + + - + + \ldots$".)
- The last three columns list, respectively, the sets of urgent, contending, and tnegru tasks at each time. As defined in Section 4, the urgent tasks at any time t are those with a strictly positive lag and a $+$ or 0 in position t of the characteristic string, while the tnegru tasks are those with a strictly negative lag and a $-$ or a 0 in position t of the characteristic string. All remaining tasks are contending. The contending tasks are listed in Table 2 by order of priority, according to the total order $\succeq$ defined in Section 4. More specifically, an expression of the form $(x > y)$ in Table 2 indicates that $(x \succeq y)$ holds, but $(y \succeq x)$ does not, while an entry of the form $(x = y)$ indicates that both $(x \succeq y)$ and $(y \succeq x)$ hold.

The schedule generated by Algorithm PF can easily be determined from Table 2. (Below we describe how Algorithm PF computes the table entries.) At each time t, all of the urgent

t	lag × period					characteristic string					urgent tasks	contending tasks	tnegru tasks
	v	w	x	y	z	v	w	x	y	z			
0	0	0	0	0	0	−	−	−	−	−	$\{\}$	$y > z > x > w > v$	$\{\}$
1	1	2	−2	−3	−127	−	0	+	+	+	$\{w\}$	$y > z > x > v$	$\{\}$
2	2	0	3	−6	−254	0	−	+	+	+	$\{v, x\}$	$w > y > z$	$\{\}$
3	0	−2	1	2	81	−	0	−	−	−	$\{\}$	$y > z > x > v$	$\{w\}$
4	1	0	−1	−1	−46	−	−	+	+	+	$\{\}$	$y > z > x > v = w$	$\{\}$
5	2	2	−3	−4	−173	0	0	+	+	+	$\{v, w\}$	$y > z > x$	$\{\}$
6	0	0	2	−7	162	−	−	0	+	+	$\{x, z\}$	$w > y > v$	$\{\}$
7	1	−2	0	1	35	−	0	−	−	−	$\{\}$	$y > z > x > v$	$\{w\}$
8	2	0	−2	−2	−92	0	−	+	+	+	$\{v\}$	$y > z > x > w$	$\{\}$
9	0	2	3	−5	−219	−	0	+	+	+	$\{w, x\}$	$y > z > v$	$\{\}$
10	1	0	1	−8	116	−	−	−	0	−	$\{\}$	$z > x > v = w$	$\{y\}$
11	−1	2	−1	0	−11	0	0	+	−	+	$\{w\}$	$y > z > x$	$\{v\}$
12	0	0	4	−3	−138	−	−	+	+	+	$\{x\}$	$y > z > w > v$	$\{\}$
13	1	2	2	−6	−265	−	0	0	+	+	$\{w, x\}$	$v > y > z$	$\{\}$
14	−1	0	0	2	70	0	−	−	−	−	$\{\}$	$y > z > x > w$	$\{v\}$
15	0	2	−2	−1	−57	−	0	+	+	+	$\{w\}$	$y > z > x > v$	$\{\}$
16	1	0	3	−4	−184	−	−	+	+	+	$\{x\}$	$y > z > v = w$	$\{\}$
17	2	2	1	−7	−311	0	0	−	+	+	$\{v, w\}$	$x > y > z$	$\{\}$
18	0	0	−1	1	24	−	−	+	−	−	$\{\}$	$y > z > x > w > v$	$\{\}$
19	1	2	−3	−2	−103	−	0	+	+	+	$\{w\}$	$y > z > v = x$	$\{\}$

Table 2: An example execution of Algorithm PF.

tasks, and none of the tnegru tasks, are allocated copies of the resource. Any remaining resources are allocated to the highest-priority contending tasks, with ties broken arbitrarily. (Theorem 2 establishes that such a schedule is always possible by proving that: (i) the number of urgent tasks does not exceed the number of resources, and (ii) the number of non-tnegru tasks is greater than or equal to the number of resources.) At time 2, for example, the three copies of the resource are allocated to tasks v, x, and w; similarly, at time 14, the resources are allocated to tasks y, z, and x.

Before proceeding to describe how Algorithm PF computes the entries in Table 2, we remark that the schedule of Table 2 is easily seen to be P-fair. To verify this claim, observe that all lags in the table lie in the real interval $(-1, +1)$, that is, the absolute value of each "lag $\times$ period" entry is strictly smaller than the corresponding period.

At time 0, the lags of all tasks are, by definition, zero; furthermore, all tasks are contending. The total ordering of the tasks according to the relation $\succeq$ is as given in the next-to-last column of Table 2. This is determined by lexicographically comparing the characteristic substrings of the tasks, with $+$ having priority over 0, and 0 over $-$. (For example, in comparing tasks x and w, the characteristic substrings $\alpha(x, 0) =$ "$++-++0$" and $\alpha(w, 0) =$ "0" are compared, and task x is determined to have greater priority.) Algorithm PF therefore allocates the three resources to tasks y, z, and x. The lags for time 1 are now computed as follows. For each task u that was allocated a resource at time 0, the quantity "lag $\times$ period" is decremented by $(u.p - u.e)$. (This corresponds to decrementing the lag by $(1 - u.w)$.) For each task u that was not allocated a copy of the resource at time 0, the quantity "lag $\times$ period" is incremented by $u.e$. (This corresponds to increasing the lag by $u.w$). Given these lags at time 1, Algorithm PF now determines which three tasks to schedule in slot 1 (the urgent task w and the highest-priority contending tasks y and z are selected), computes the lags corresponding to time 2, and so on.

Note that Algorithm PF does not need to store the entire schedule prior to time t in order to allocate the resources at time t. In fact, the schedule at time t can be determined from the set of task execution requirements, periods, and lags at time t. The space required by Algorithm PF is therefore linear in the size of the input instance.

We conclude our example by discussing how, given the lags at time 11, Algorithm PF allocates the three resources at time 12 and updates Table 2. At time 11, task w is urgent since it has a positive lag and a 0 in its characteristic string. Similarly, task v is tnegru since it has a negative lag and a 0 in its characteristic string. The remaining three tasks are contending, and, by comparing the characteristic substrings $\alpha(x, 11) =$ "$+0$", $\alpha(y, 11) =$ "$++-+++\ldots$", and $\alpha(z, 11) =$ "$++-+++\ldots$", we find that task y has the highest priority, followed by z and x in that order. (Observe that the entries in Table 2 are not, in themselves, sufficient to determine that $y \succeq z$, since the characteristic substrings are tied in the first eight places.) Algorithm PF therefore allocates the three resources to tasks w, y, and z. The lags are now updated in the manner discussed earlier; for example, the "lag $\times$ period" entry of task w for time 12 is decreased by $(w.p - w.e)$ to $(2 - (4 - 2)) = 0$, while that of task x is increased by $x.e$ to $(-1 + 5) = 4$.

6 The Comparison Algorithm

We will now present two implementations of the characteristic substring comparison function required by Algorithm PF. The first, which we call NaiveCompare, we prove correct. The second, Compare, we prove equivalent to the first and show that it runs in polynomial time. Both subroutines use only integer variables, and the integer operations $\{-, +, \cdot, \mathrm{mod}\}$. We will prove that the number of integer operations performed by Compare on tasks x and y is at most linear in the size of the binary representation of $\min\{x.p, y.p\}$. (Furthermore, all intermediate values can be represented in $\lceil \lg(\max\{x.p, y.p\}) \rceil$ bits.)

Subroutine Compare can be used as the basis for an implementation of Algorithm PF that requires at most linear time (in the size of instance Φ) to decide which m-subset of the n tasks to schedule in a given slot. A detailed sketch of this linear-time implementation is given in Section 6.3.

6.1 A Naive Implementation

This subsection presents a naive implementation of the characteristic substring comparison algorithm. Given contending tasks x and y at time t, our goal is to determine whether: (i) $\alpha(x,t) < \alpha(y,t)$, (ii) $\alpha(x,t) > \alpha(y,t)$, or (iii) $\alpha(x,t) = \alpha(y,t)$. The naive approach is to compare the two substrings one symbol at a time. Note that for any P-fair schedule S and $i \in [0, |\alpha(x,t)|)$:

$$\begin{aligned}
\alpha_i(x,t) &= \alpha_{t+i+1}(x) \\
&= \mathrm{sign}(x.w \cdot (t+i+2) - \lfloor x.w \cdot (t+i+1) \rfloor - 1) \\
&= \mathrm{sign}(\mathsf{lag}(S,x,t) + x.w \cdot (i+2) - \lfloor \mathsf{lag}(S,x,t) + x.w \cdot (i+1) \rfloor - 1) \\
&= \mathrm{sign}(x.p \cdot \mathsf{lag}(S,x,t) + x.e \cdot (i+2) \\
&\qquad - x.p \cdot \lfloor (x.p \cdot \mathsf{lag}(S,x,t) + x.e \cdot (i+1))/x.p \rfloor - x.p) \\
&= \mathrm{sign}(x.e - x.p + (x.p \cdot \mathsf{lag}(S,x,t) + x.e \cdot (i+1)) \bmod x.p),
\end{aligned}$$

where the last equation follows from the identity $a \cdot \lfloor b/a \rfloor = b - b \bmod a$, for positive integers a and b. If task x is contending at time t under P-fair schedule S, we have $(x.p \cdot \mathsf{lag}(S,x,t) + x.e) \in (0, x.p)$. Hence

$$\alpha_0(x,t) = \mathrm{sign}(x.p \cdot \mathsf{lag}(S,x,t) + 2 \cdot x.e - x.p).$$

Let

$$\begin{aligned}
a_0 &\stackrel{\text{def}}{=} x.p - x.e, \\
b_0 &\stackrel{\text{def}}{=} x.e, \text{ and} \\
c_0 &\stackrel{\text{def}}{=} x.p \cdot \mathsf{lag}(S,x,t) + 2 \cdot x.e - x.p.
\end{aligned}$$

Note that $a_0 \in (0, x.p)$, $b_0 \in (0, x.p)$, and $c_0 \in (-a_0, b_0)$. Define a_1, b_1, and c_1 similarly with respect to task y. Given a_0, b_0, and c_0, it is straightforward to compute $\alpha(x,t)$ one symbol at a time, using a constant number of integer operations per symbol. Of course, $\alpha(y,t)$ can be computed in a similar fashion. This is the approach taken in subroutine NaiveCompare below. Note that in the ith iteration of the **do** loop, we have $\text{sign}(c_0) = \alpha_i(x,t)$ and $\text{sign}(c_1) = \alpha(y,t)$.

```
NaiveCompare(a0, b0, c0, a1, b1, c1)
int a0, b0, c0, a1, b1, c1;
{
   do c0 > 0 ∧ c1 > 0 ⟶ c0, c1 := c0 − a0, c1 − a1
    [] c0 < 0 ∧ c1 < 0 ⟶ c0, c1 := c0 + b0, c1 + b1
   od;
   if c0 = 0 ∧ c1 = 0 ⟶ return TIE fi;
   if c0 ≥ 0 ∧ c1 ≤ 0 ⟶ return 0
    [] c0 ≤ 0 ∧ c1 ≥ 0 ⟶ return 1
   fi
}
```

The return values of NaiveCompare are 0, 1, and TIE. The return value 0 indicates that the task corresponding to the triple (a_0, b_0, c_0) should be given priority over the one corresponding to the triple (a_1, b_1, c_1). Conversely, the return value 1 indicates that the triple (a_1, b_1, c_1) should have priority. The return value TIE indicates that either can be scheduled ahead of the other. As mentioned in Section 4, such a tie could be broken using the task numbers.

Definition 6.1 A triple (a, b, c) is *admissible* if and only if: (i) a and b are positive integers, and (ii) c is an integer in the interval $(-a, b)$ such that $\gcd\{a, b\} \mid c$. We will say that a 6-tuple $(a_0, b_0, c_0, a_1, b_1, c_1)$ is *admissible* if and only if (a_0, b_0, c_0) and (a_1, b_1, c_1) are admissible triples.

It is immediate from the foregoing discussion that every input 6-tuple passed to NaiveCompare by our scheduling algorithm is admissible. Condition (ii) implies that NaiveCompare will eventually terminate. Unfortunately, the running time of NaiveCompare is not very good; it is pseudo-polynomial in the input size. This deficiency will be addressed in the next section.

6.2 An Efficient Implementation

In this section, we present a polynomial-time subroutine Compare with the same input-output behavior as the NaiveCompare subroutine of Section 6.1. The algorithm is recursive. As argued in Section 6.1, we can assume that any 6-tuple of arguments passed to the NaiveCompare subroutine is admissible. Correspondingly, the arguments of any top-level call to Compare may be assumed to be admissible. Lemma 6.1 below proves that this assumption can be extended to any non-trivial depth of recursion.

Lemma 6.1 If algorithm Compare is called with an admissible 6-tuple, then every resulting recursive call will also involve an admissible 6-tuple.

Proof: Assume that Compare is called with admissible 6-tuple $(a_0, b_0, c_0, a_1, b_1, c_1)$. Note that for $0 \le i \le 1$, a_i and b_i are not changed within Compare but that c_i is assigned a new value at Line 8. For the sake of clarity, let C_i represent the value passed to c_i in the call to Compare and let C_i' represent the value of c_i after Line 8. To prove the lemma we will establish the following pair of claims, for $0 \le i \le 1$: (i) if the recursive call in Line 4 of Compare is executed then $(b_i, a_i, -C_i)$ is an admissible triple, and (ii) if the recursive call in Line 16 is executed then (a_i', b_i', c_i'), defined as

$$(a_i - (b_i \bmod a_i),\ b_i \bmod a_i,\ C_i' + (b_i \bmod a_i)),$$

is an admissible triple. The proof of Claim (i) is straightforward; (a_i, b_i, C_i) is admissible if and only if $(b_i, a_i, -C_i)$ is admissible.

We now address Claim (ii). First, note that if $a_i \mid C_i$ then the recursive call at Line 16 is not reached. Thus we can assume that $a_i \not| C_i$, which easily implies $a_i \not| b_i$ and $a_i \not| C_i'$. Line 8 sets C_i' to $-a_i + (C_i \bmod a_i)$ and hence $c_i' = -a_i + (C_i \bmod a_i) + (b_i \bmod a_i)$. If $a_0 \ge b_0$ or $a_1 \ge b_1$ then again the recursive call at Line 16 is not reached. Thus we can assume that $(b_i \bmod a_i) \in (0, a_i)$ and both $a_i' = (a_i - (b_i \bmod a_i))$ and $b_i' = (b_i \bmod a_i)$ are positive integers. It remains to prove that $\gcd\{a_i', b_i'\} \mid c_i'$ and that $c_i' \in (-a_i', b_i')$.

The identities $\gcd\{m, n\} = \gcd\{m, m-n\}$ and $\gcd\{m, n\} = \gcd\{m, m \bmod n\}, m > n > 0$, are easily verified. (Note that two common versions of Euclid's GCD algorithm depend on these identities.) The second identity implies that $\gcd\{a_i, b_i'\} = \gcd\{a_i, b_i\}$. The first identity implies that $\gcd\{a_i', b_i'\} = \gcd\{a_i, b_i'\}$ and therefore $\gcd\{a_i', b_i'\} = \gcd\{a_i, b_i\}$. For convenience, let $g_i = \gcd\{a_i, b_i\} = \gcd\{a_i', b_i'\}$. Because (a_i, b_i, C_i) is an admissible triple, $g_i \mid C_i$. Since $g_i \mid a_i$ we have $g_i \mid (C_i \bmod a_i)$. Note that $(b_i \bmod a_i) = b_i'$, and so $g_i \mid (b_i \bmod a_i)$. Thus, $c_i' = -a + (C_i \bmod a_i) + (b_i \bmod a_i)$ is a sum of multiples of g_i and therefore is itself a multiple of g_i.

Finally, because $(b_i \bmod a_i)$ and $(C_i \bmod a_i)$ are both in $(0, a_i)$, it follows that $-a_i + (b_i \bmod a_i) < -a_i + (C_i \bmod a_i) + (b_i \bmod a_i) < (b_i \bmod a_i)$. Hence, $c_i' \in (-a_i', b_i')$, completing the proof of Claim (ii). □

Theorem 3 Let $d = \min\{\ell(a_0), \ell(b_0), \ell(a_1), \ell(b_1)\}$ where $\ell(i) = \lfloor \lg(i+1) \rfloor$. Then algorithm Compare performs $O(d)$ integer operations.

Proof: Since algorithm Compare does not contain any loops and uses only tail recursion, it is sufficient to prove that the maximum depth of recursion is $O(d)$. More precisely, we will prove by induction that the maximum depth of recursion is $2d-2$ if $\min\{a_0, a_1\} \le \min\{b_0, b_1\}$, and $2d-1$ otherwise.

The base of our induction is $d = 1$. (By Lemma 4.1, $d > 0$.) Note that if (a, b, c) is an admissible triple, then $a + b \ge 2$. Thus, using Lemma 6.1, we have $a_i + b_i \ge 2$, $0 \le i \le 1$. If $d = 1$ then $a_0 = a_1 = b_0 = b_1 = 1$, which implies $c_0 = c_1 = 0$. Thus the depth of recursion is $2d - 2 = 0$, as claimed.

For the induction step, assume that $d \ge 2$ and that the claim holds for smaller values of d. We consider two cases:

1. If $\min\{a_0, a_1\} \le \min\{b_0, b_1\}$ (and the depth of recursion is greater than 0), Line 16 of Compare must be executed. Let a_i', b_i', and c_i' be defined as in the proof of Lemma 6.1,

and assume without loss of generality that $a_0 \le a_1$. Thus, $\ell(a_0) = d \ge 2$. Since $a_0' + b_0' = a_0$, $\min\{\ell(a_0'), \ell(b_0')\} < d$. The claim then follows by the induction hypothesis.

2. If $\min\{a_0, a_1\} > \min\{b_0, b_1\}$ then the recursive call in Line 4 of Compare will be executed. That call will terminate within at most $2d - 2$ additional levels of recursion by the argument of the preceding case. Thus, the maximum depth of recursion is at most $2d - 1$, as claimed.

□

(1) Compare($a_0, b_0, c_0, a_1, b_1, c_1$)
(2) **int** $a_0, b_0, c_0, a_1, b_1, c_1$;
(3) {
(4)   **if** $\min\{a_0, a_1\} > \min\{b_0, b_1\} \longrightarrow$ **return** Compare($b_1, a_1, -c_1, b_0, a_0, -c_0$) **fi**;
(5)   **if** $\lceil c_0/a_0 \rceil > \lceil c_1/a_1 \rceil \longrightarrow$ **return** 0
(6)   [] $\lceil c_0/a_0 \rceil < \lceil c_1/a_1 \rceil \longrightarrow$ **return** 1
(7)   **fi**;
(8) $c_0, c_1 := c_0 - a_0 \cdot \lceil c_0/a_0 \rceil,\ c_1 - a_1 \cdot \lceil c_1/a_1 \rceil$;
(9)   **if** $c_0 = 0 \wedge c_1 = 0 \longrightarrow$ **return** TIE
(10)   [] $c_0 \neq 0 \wedge c_1 = 0 \longrightarrow$ **return** 0
(11)   [] $c_0 = 0 \wedge c_1 \neq 0 \longrightarrow$ **return** 1
(12)   **fi**;
(13)   **if** $\lfloor b_0/a_0 \rfloor > \lfloor b_1/a_1 \rfloor \longrightarrow$ **return** 0
(14)   [] $\lfloor b_0/a_0 \rfloor < \lfloor b_1/a_1 \rfloor \longrightarrow$ **return** 1
(15)   **fi**;
(16)   **return** Compare($a_0 - (b_0 \bmod a_0),\ b_0 \bmod a_0,\ c_0 + (b_0 \bmod a_0),$
(17) $a_1 - (b_1 \bmod a_1),\ b_1 \bmod a_1,\ c_1 + (b_1 \bmod a_1)$)
(18) }

It remains to argue that: (i) Compare never executes a division by 0, and (ii) Compare always returns the correct value. Claim (i) is easy to justify: all divisions are by a_0 or a_1, which are strictly positive. Claim (ii) is addressed by the following theorem.

Theorem 4 On any admissible input 6-tuple, algorithms NaiveCompare and Compare return the same value.

Proof: In the following, let σ_i denote the characteristic substring associated with the admissible triple (a_i, b_i, c_i), $0 \le i \le 1$.

We will prove the theorem by induction on the depth of recursion used by algorithm Compare. By Theorem 3, this depth is finite. For the base case, assume that Compare does not call itself recursively, i.e., that the maximum depth of recursion is 0. Thus, one of the non-recursive **return** statements is executed (the two recursive **return** statements are in Lines 4 and 16). In the argument that follows we will deal with each of the non-recursive **return** statements in turn.

Since the recursive call on Line 4 is not executed, we can assume that $\min\{a_0, a_1\} \le \min\{b_0, b_1\}$. Now consider the two quantities, $\lceil c_0/a_0 \rceil$ and $\lceil c_1/a_1 \rceil$, being compared in Lines 5 and 6. Note that the string σ_i must begin with $\lceil c_i/a_i \rceil$ +'s, followed by either a $-$ or a 0. Thus, the **return** statements of Lines 5 and 6 correctly handle any case where $\lceil c_0/a_0 \rceil \neq \lceil c_1/a_1 \rceil$.

If execution proceeds beyond Line 7, let $t = \lceil c_0/a_0 \rceil (= \lceil c_1/a_1 \rceil)$. Note that Line 8 then sets c_0 and c_1 to the values these variables would have attained in NaiveCompare after processing the common prefix of t +'s in σ_0 and σ_1 (i.e., after exiting the **do** loop). Let σ_i' denote the string σ_i with this common prefix removed, $0 \leq i \leq 1$. It remains to compare strings σ_0' and σ_1'.

Note that after executing Line 8, we have $c_i \in (-a_i, 0]$, $0 \leq i \leq 1$. If either c_0 or c_1 is equal to 0, we can immediately determine the outcome of the comparison between strings σ_0' and σ_1'. For example, if $c_0 = 0$ and $c_1 \neq 0$ then $\sigma_0' > \sigma_1'$ because $\sigma_0 = +^t 0$, whereas the first $t+1$ symbols of the string σ_1 are $+^t-$. Reasoning in this manner, we can see that the three **return** statements of Lines 9, 10, and 11 correctly handle any case where either c_0 or c_1 is equal to 0.

If execution proceeds beyond Line 12, we have $c_i \in (-a_i, 0)$, $0 \leq i \leq 1$. For each i, $0 \leq i \leq 1$, we now consider three cases:

- Case 1: $c_i = -(b_i \bmod a_i)$. In this case, it is easy to verify that $\sigma_i' = - +^{\lfloor b_i/a_i \rfloor} 0$. In what follows, let $\odot_i$ denote the string $- +^{\lfloor b_i/a_i \rfloor} 0$.
- Case 2: $c_i \in (-a_i, -(b_i \bmod a_i))$. In this case, the first $\lfloor b_i/a_i \rfloor + 1$ symbols of σ_i' form the string $\ominus_i \stackrel{\text{def}}{=} -+^{\lfloor b_i/a_i \rfloor}$. Let c_i' denote the new value of c_i after processing these symbols as in NaiveCompare. Then

$$\begin{aligned} c_i' &= c_i + b_i - a_i \cdot \lfloor b_i/a_i \rfloor \\ &= c_i + (b_i \bmod a_i). \end{aligned}$$

  Note that $c_i' \in (-a_i, 0)$.
- Case 3: $c_i \in (-(b_i \bmod a_i), 0)$. In this case, the first $\lceil b_i/a_i \rceil + 1$ symbols of σ_i' form the string $\oplus_i \stackrel{\text{def}}{=} -+^{\lceil b_i/a_i \rceil}$. Let c_i' denote the new value of c_i after processing these symbols as in NaiveCompare. Then

$$\begin{aligned} c_i' &= c_i + b_i - a_i \cdot \lceil b_i/a_i \rceil \\ &= c_i - (a_i - (b_i \bmod a_i)). \end{aligned}$$

  Note that $c_i' \in (-a_i, 0)$.

In Case 1 above, we completely characterize the string σ_i'. In Cases 2 and 3, we identify a prefix of σ_i' and find that after processing that prefix, the new value of c_i remains in the interval $(-a_i, 0)$, meaning that the preceding case analysis can be repeated on the remaining suffix of σ_i'. In other words, the string σ_i' may be viewed as a sequence of $\ominus_i$'s and $\oplus_i$'s, followed by a single occurrence of $\odot_i$. Whenever $\lfloor b_0/a_0 \rfloor \neq \lfloor b_1/a_1 \rfloor$, we can immediately determine which of the strings σ_0' and σ_1' is lexicographically greater. In particular, the **return** statements of Lines 13 and 14 correctly handle any case in which $\lfloor b_0/a_0 \rfloor \neq \lfloor b_1/a_1 \rfloor$.

We have now completed the base case of the induction, that is, we have proven that Compare works correctly (i.e., returns the same value as NaiveCompare) on any admissible input for which no recursive call is generated. It remains to consider the induction step. Accordingly, let us assume that algorithm Compare works correctly on any admissible input leading to a maximum depth of recursion strictly less than d, $d > 0$. It remains to prove that Compare works correctly on any admissible input $(a_0, b_0, c_0, a_1, b_1, c_1)$ with associated maximum depth of recursion $d > 0$. There are two cases to be considered: (i) the top-level recursive call is made in Line 4, and (ii) the top-level recursive call is made in Line 16.

The case in which the top-level recursive call occurs in Line 4 is quite easy to handle. Let τ_i denote the characteristic substring associated with the admissible triple $(b_i, a_i, -c_i)$, $0 \le i \le 1$. Note that the strings σ_i and τ_i are closely related. In particular, they are "complementary" strings in the sense that one can be obtained from the other by changing $-$'s to $+$'s, $+$'s to $-$'s, and leaving the 0 symbol unchanged. With this observation, it is easy to see that NaiveCompare will return the same result on $(b_1, a_1, -c_1, b_0, a_0, -c_0)$ as it would on $(a_0, b_0, c_0, a_1, b_1, c_1)$. By the induction hypothesis, the recursive call of Line 4 will function correctly, completing the analysis of this case.

It remains to consider the case in which the top-level recursive call occurs in Line 16. Our base case analysis implies that when Line 16 is executed: (i) $c_0 \in (-a_0, 0)$, (ii) $c_1 \in (-a_1, 0)$, (iii) $\ominus \stackrel{\text{def}}{=} \ominus_0 = \ominus_1$, (iv) $\oplus \stackrel{\text{def}}{=} \oplus_0 = \oplus_1$, (v) $\odot \stackrel{\text{def}}{=} \odot_0 = \odot_1$, (vi) strings σ'_0 and σ'_1 (as defined in the base case analysis) can be viewed as strings of $\ominus$'s and $\oplus$'s terminated by a $\odot$.

Furthermore, it is straightforward to prove that σ'_i, viewed as a string over $\{\ominus, \odot, \oplus\}$, corresponds to the characteristic substring of the admissible triple

$$(a_i - (b_i \bmod a_i),\ b_i \bmod a_i,\ c_i + (b_i \bmod a_i)),$$

$0 \le i \le 1$. (To make the correspondence, replace $\ominus$ by $-$, $\odot$ by 0, and $\oplus$ by $+$.) Thus, the induction hypothesis implies that the recursive call of Line 16 correctly compares strings σ'_0 and σ'_1. □

6.3 A Linear-Time Implementation of Algorithm PF

In this section we describe how subroutine Compare can be used as the basis for a linear-time (in the size of instance Φ) implementation of Algorithm PF.

A single call to subroutine Compare can be used to determine the relative priority of any two contending tasks. Thus, by applying subroutine Compare within any optimal comparison-based selection algorithm (e.g., [2]), we can obtain an implementation of Algorithm PF that makes $O(n)$ calls to subroutine Compare to decide which m-subset of the n tasks to schedule in any given slot. This simple approach yields a polynomial-time scheduling algorithm, but unfortunately does not yield the desired linear-time bound. The problem is that the cost of individual calls to subroutine Compare can vary widely, since the cost depends on the number of bits in the arguments passed to Compare. For example, if a significant fraction of the $O(n)$ calls to Compare happen to involve pairs of tasks with a substantially greater-than-average number of bits in the binary representations of their periods, then the overall running time of this implementation of Algorithm PF could be super-linear. In the remainder of this section, we sketch the details of a slightly more complicated implementation of Algorithm PF that achieves the desired linear-time bound. Our approach is based on generalizing subroutine Compare to examine the entire set of contending tasks at once, rather than two at a time.

Note that subroutines NaiveCompare and Compare both have the following high-level structure: For two given input tasks, successive "tie-breakers" are applied (i.e., an integer is calculated for each task and these integers are compared) until either the tie is broken (i.e., the relative priority of the two tasks has been determined) or it is determined that the two tasks have identical characteristic substrings (i.e., the two tasks have equal priority). Furthermore, the complexity of either of these subroutines is given by the worst-case length of the sequence of tie-breakers, since each tie-breaker requires only a constant number of integer operations. In the case of subroutine Compare, Theorem 3 implies that the length of the sequence of tie-breakers is $O(d)$, where d is as defined in the statement of the theorem. (For the purposes of the present analysis, it is sufficient to observe that d is no larger than the minimum number of bits in the binary representations of the two task periods.)

Assume without loss of generality that: (i) we are in the process of scheduling slot t, $t \geq 0$, (ii) $m' \leq m$ tasks are urgent at time t, and (iii) $n' \leq n - m'$ tasks are contending at time t. Thus, it remains to select the highest-priority $(m-m')$-subset of the n' contending tasks. We now describe a generalized version of subroutine Compare to accomplish this objective. The generalized subroutine works by applying a sequence of tie-breakers to successive subsets $C_0 \supseteq \cdots \supseteq C_k = \emptyset$ of the set of contending tasks. (These tie-breakers correspond to the tie-breakers performed by the two-task version of subroutine Compare.)

Tie-breaker 0 is applied to the entire set of contending tasks (i.e., C_0 is the set of contending tasks). In general, tie-breaker i behaves as follows. First, an integer is calculated for each task in C_i. (The integer associated with a given task x in C_i is the same as the integer that would be assigned to x by tie-breaker i in a call to the two-task version of Compare involving x and any other task y in C_i.) Second, a linear-comparison selection algorithm is applied to this set of integers to partition C_i into the following three uniquely-determined subsets: (i) the subset C_i' of tasks which must be scheduled in slot t on the basis of tie-breaker i, (ii) the subset C_i'' of tasks which must not be scheduled in slot t on the basis of tie-breaker i, and (iii) the remaining contending tasks C_{i+1}. Note that every task in C_{i+1} has the same associated integer in tie-breaker i (as well as in all previous tie-breakers).

The preceding discussion can be formalized in a straightforward fashion by using induction over the sequence of tie-breakers. Now observe that a given task x cannot belong to C_i unless i is Big-Oh of the number of bits in the binary representation of $x.p$. The linear-time bound follows immediately.

7 Conclusions

We have defined a new notion of fairness, called P-fairness, which we believe to be quite useful in a variety of resource allocation problems. We have shown that P-fair schedules exist for the resource sharing problem, which is a slight generalization of the periodic scheduling problem. Furthermore, we have provided an efficient algorithm for computing a P-fair schedule.

The swapping argument of Lemma 4.6 captures the essence of P-fairness by modeling exchanges that are permissible in P-fair schedules. An interesting problem for future research is to identify generalizations of the periodic scheduling problem that can be handled within the same framework.

The Compare subroutine appears to be closely related to Euclid's GCD algorithm, as well as to various algorithms that have been proposed for 2-ILP, that is, integer linear programming with two variables [6, 8, 13, 14]. (ILP is NP-complete in general, but can be solved in polynomial-time for any fixed number of variables [9].) Deng has extensively studied the relationship between GCD and 2-ILP [3].

Our P-fair scheduling algorithm produces schedules with a large number of preemptions. It would be interesting to investigate algorithms for solving the periodic scheduling problem which minimize the number of preemptions.

8 Acknowledgments

We would like to thank A. K. Mok and C. L. Liu for their encouragement and interest in this problem. We are indebted to the members of the Austin Tuesday Afternoon Club for their many helpful comments and suggestions. In particular, E. W. Dijkstra suggested a nice simplification of the NaiveCompare subroutine. We would also like to thank the anonymous referees for their comments, which have helped to improve the readability of the paper.

References

[1] S. K. Baruah, R. R. Howell, and L. E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.

[2] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *JCSS*, 7:448–461, 1973.

[3] X. Deng. *Mathematical Programming: Complexity and Applications*. PhD thesis, Department of Operations Research, Stanford University, Stanford, CA, September 1989.

[4] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, NY, 1979.

[6] D. S. Hirschberg and C. K. Wong. A polynomial-time algorithm for the knapsack problem with two variables. *JACM*, 23:147–154, 1976.

[7] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[8] R. Kannan. A polynomial algorithm for the two-variable integer programming problem. *JACM*, 27:118–122, 1980.

[9] H. W. Lenstra, Jr. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.

[10] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4:209–219, 1989.

[11] C. L. Liu. Scheduling algorithms for multiprocessors in a hard-real-time environment. JPL Space Programs Summary 37–60, vol. II, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, pages 28–37, November 1969.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20:46–61, 1973.

[13] H. E. Scarf. Production sets with indivisibilities, Part I: Generalities. *Econometrica*, 49:1–32, 1981.

[14] H. E. Scarf. Production sets with indivisibilities, Part II: The case of two activities. *Econometrica*, 49:395–423, 1981.