A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems: A Closer Look

Sarah Barnes Haddix

April 2026

Abstract

Now that the *Earlist Virtual Deadline First* algorithm, or EEVDF, is the default fair scheduler in the Linux kernel, we present expanded examples and explanations for its core concepts and a discussion of the current implementation of the algorithm in Linux. Further, we provide revised and expanded proofs bounding the error of the schedule produced by EEVDF with respect to an ideal proportional-share allocation and proving the optimality of these bounds in the class of proportional-share allocation algorithms.

1 Introduction

The EEVDF algorithm, first proposed in 1995 by Stoica and Abdel-Wahab [1] and further developed in 1996 by Stocia et al. [2], is a scheduling algorithm that is flexible enough to both promote fairness and ensure bounded latency. It is well-suited for handling dynamic task systems: those in which tasks leave and join unpredictably and whose execution time has wide variations. EEVDF treats fairness as a first-class concept. By this, we mean that the algorithm was designed to ensure fairness and we derive timeliness guarantees from that definition, rather than vice versa, as is often the case, particularly with real-time scheduling algorithms. In fact, under EEVDF, real and non-real-time processes are treated identically; they are scheduled in the same system according to exactly the same set of rules.

The EEVDF algorithm is a proportional-share algorithm, which means that it allocates resources to a task based on its weighted share of the system. That is, if a task is prescribed to have double the share of another task, the scheduler will endeavor to allocate it double the time on the resource.

The EEVDF algorithm implements proportional-share allocation by defining an ideal schedule, a theoretical schedule that achieves perfect fairness by having tasks make uniform proportional progress. The scheduling algorithm is defined

by attempting to minimize error with respect to this ideal schedule, a concept known as lag. These concepts are formalized and expanded on in Section 2.

In this work, we revisit the EEVDF algorithm as presented in [2] and in [1], providing a discussion of the current state of EEVDF in the Linux operating system and revised proofs of the results relevant to this modern implementation. Further, we provide a novel proof that EEVDF is work-conserving, found in the Appendix.

Section 2 provides a complete description of the EEVDF algorithm for non-dynamic systems. Section 3 presents related work on fair scheduling algorithms. Section 4 provides a complete description of the EEVDF algorithm for dynamic systems. Section 5 presents the basics of the current EEVDF algorithm in the Linux kernel. Section 6 provides a discussion on the use of EEVDF in real-time contexts. Section 7 provides an overview of the mathematical results established in [2] and the changes made to their proofs in this work. Section 8 concludes the paper.

2 The Model and Algorithm

2.1 The Task Model

In this work, we assume the same task model introduced in [2]. We consider *clients* that issue *requests* for time with exclusive access to some resource (*e.g.*, the CPU). A client is an abstraction used in the same way as an operating system process or a task in real-time systems. A request is a schedulable entity. We use the notation τ_i to refer to client i and r_i^k to its k^{th} request. We use the term *task* interchangeably with client.

A *pending* request is a request that has been issued by a client, but not yet fulfilled. A request is fulfilled when it has received its requested time on the resource. A client is *active* if it has a pending request and *passive* otherwise. We allow each client to have at most one pending request at a time. For this reason, the notions of scheduling a client and scheduling a client's request are equivalent and both wordings are used interchangeably in this work.

A task is said to leave the system when it is no longer active after having been and is said to join the system when it becomes active after not having been, or when issuing its first request. These notions are made more precise in Section 4. We will often talk about the system as a *competition*, *i.e.,* a client leaves or joins the competition when it leaves or joins the system. In this work, we will think of all active clients as competing for the CPU.

For this analysis, we will assume continuous time, *i.e.,* that between two units of time always exists another.

In this analysis, the CPU is allocated to requests in time quanta of at most length q. This means that when the scheduler makes a scheduling decision, it will allocate at most q time units to its selected client. These decisions are not necessarily made at uniform intervals. Take, for example, the case where a client is allocated a full quantum but finishes early. The scheduler is invoked and a new

client τ_i is allocated the CPU, again for at most q time units. This scheduling decision is made at a time that is not an integer multiple of the quantum size q and varies depending on the time that τ_i finishes. These concepts respectively are called *fractional* and *non-uniform time quanta* and we will see how the EEVDF algorithm accounts for them in Section 4.

Each client τ_i in the competition is assigned a weight $w_i \in \mathbb{Q}$, where $w_i > 0$, which determines the share of time on the CPU it will have. Each active client's instantaneous *share* at time t is defined by

$$f_i(t) = \frac{w_i}{\sum_{j \in A(t)} w_j}\,, \tag{1}$$

where the sum is over $A(t)$, the set of active clients in the competition at time t. A passive client has a weight of 0 and therefore a share of 0.

Now, we introduce the *fluid schedule*, or *ideal schedule*, used interchangeably. The fluid schedule is an idealized schedule in which requests make perfect uniform progress at a rate proportional to their share of the system. In other words, one can take any time interval of any size, and over that interval each client will have been allocated time proportional to its share. This schedule is not physically realizable; it is achieved by allocating the CPU in infinitesimally small time quanta. As an example, consider a system with four clients of equal weight and thus equal share. In an arbitrary time interval of length ℓ, the ideal schedule will have allocated each client exactly $\frac{\ell}{4}$ time units. We define the time allocated to τ_i in the ideal system during the interval $[t_1, t_2)$ as

$$S_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(t)\mathrm{d}t\,. \tag{2}$$

We note that for all active τ_i, by (1) and $w_j > 0$ for all active clients

$$0 < f_i(t) \leq 1\,, \tag{3}$$

and further, by (2),

$$S_i(t_1, t_2) > 0\,, \text{ for } t_1 < t_2\,. \tag{4}$$

For an interval where the weights of all clients remain constant, by combining (1) and (2), we can express the ideal allocation as

$$S_i(t_1, t_2) = \frac{w_i}{\sum_{j \in A(t_1)} w_j}(t_2 - t_1)\,. \tag{5}$$

The integral (2) is always correct and, since clients are allocated in discrete quanta, can always be expressed as a finite sum of elements of the form in (5).

We now define the concept of lag as the difference between the time a client was to be allocated in the ideal system and the time it received in the actual system. We define the lag of τ_i at time t

$$\mathrm{lag}_i(t) = S_i(t_i^0, t) - s_i(t_i^0, t)\,, \tag{6}$$

where t_i^0 is the time τ_i first became active and $s_i(t_1, t_2)$ is the CPU time τ_i received in $[t_1, t_2)$.

A client has *positive lag* when it is lagging behind the ideal system, when $s_i(t_i^0, t) < S_i(t_i^0, t)$, implying that the actual system must "catch up" to the ideal system. A client has *negative lag* when it has executed for longer in the actual system than in the ideal, when $s_i(t_i^0, t) > S_i(t_i^0, t)$, implying that it has "outrun" the ideal system. A client has *zero lag* when the ideal system and the actual system have allocated its requests for exactly the same amount of time.

Consider the following example to better understand lag. Take a system of two clients, τ_a and τ_b, with equal weight. Suppose each client always has an active request. At time 0, they each have a lag of 0. Assume that τ_a is scheduled at time 0 for 10 time units. At time 10 in the ideal system, each client should have been serviced for 50% of the time—5 time units. So, at time 10, $\text{lag}_a(10) = 5 - 10 = -5$ and $\text{lag}_b = 5 - 0 = 5$. An important property of lag seen here, formally stated in Lemma 3 of the Appendix, is that the sum of the lags of all clients in the system is always 0.

Clearly, lag provides a way to measure the error of some actual system with respect to the ideal system. We then can use lag as a metric to measure the performance of an algorithm by considering the lag of the schedules produced by the algorithm. As we will prove in Section 7, under EEVDF, the lag for any client at any point in the produced schedule is bounded by q, the length of one time quantum. Further, we will prove that this bound is optimal within the class of proportional-share resource allocation algorithms. By bounding the lag for schedules produced by EEVDF, we obtain a formal notion of fairness and, as we will explore in Section 6, are able to consider EEVDF in certain real-time contexts.

2.2 The Virtual Time Domain

Under EEVDF, tasks are scheduled in *virtual time*. Virtual time is mapped to *real time* at a rate dependent on the weights of the clients currently active in the system. The main principle behind the virtual time domain is this: each client τ_i will be scheduled for w_i real time units in each virtual time unit. This property is attained by calculating virtual time as a function of real time and the state of the system

$$V(t) = \int_0^t \frac{1}{\sum_{j \in A(x)} w_j} dx\,, \tag{7}$$

where $A(x)$ is the set of active clients in the system at time x (just as in (5)). At any point where there are no active clients in the system, we define the integrand to be 0. This equation expresses the notion that virtual time is real time scaled by the inverse of the weights of the clients in the system. This integral functions to calculate the value of virtual time in systems where clients leave, join, or change weights only at points where they have zero lag.

In this paper, we will reason using the following definition of virtual time, which holds for all systems.

$$V(t_2) = V(t_1) + \frac{1}{\sum_{j \in A(t_1)} w_j}(t_2 - t_1)\,, \tag{8}$$

for $[t_1, t_2)$ where the weights in the system are constant. For any intervals where there are no active clients in the system, we define $\frac{1}{\sum_{j \in A(t_1)} w_j} = 0$. We define $V(0) = 0$.

We think of virtual time as a parameter that starts at 0 and ticks forward at a rate proportional to the inverse of the weights in the system. We note, however, that in the case of dynamic systems where clients may leave, join, or change weights at any time, not just at points where their lag is zero, that the value of virtual time is not necessarily strictly increasing with respect to the value of real time. This will be discussed further in Section 4.

To better understand virtual time, take the simple example of a system $\tau = \{\tau_1, \tau_2\}$ with $w_1 = 1$ and $w_2 = 2$. Consider the mappings between virtual and real time presented in Figure 1 and Figure 2.

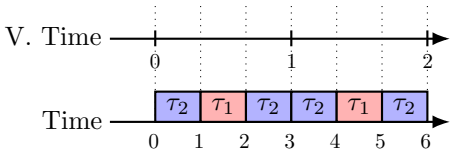

Figure 1: *Each τ_i issues its first request at $t = 0$ and issues each subsequent request as soon as the previous request is fulfilled. From (8), we find $V(1) = V(0) + \frac{1}{1+2}(1-0) = \frac{1}{3}$, which means that in $[0, 1)$, 1 virtual time unit passes every 3 time real time units, or that virtual time ticks at a rate $\frac{1}{3}$ of real time. The weights in the system do not change following $[0, 1)$ and thus neither does the slope of virtual time. We see that in each virtual time unit, there are $3 = 1 + 2 = w_1 + w_2 = \sum_{j \in A(x)} w_j$ real time units.*

In these examples, the key insight is that virtual time is defined such that each τ_i is able to execute for w_i real time units in each virtual unit.

In this work, any unqualified mention of time implies real time. Any discussion of virtual time or a duration that is measured in "virtual time units" will be made explicit.

The rate at which virtual time ticks against real time will change as clients enter and leave the system; there are three cases. First, if $\frac{1}{\sum_j w_j} = 1$, virtual time ticks at the same speed as real time; x time units in virtual-time are x time units in real time. Second, if $\frac{1}{\sum_j w_j} > 1$, virtual time ticks faster than real time. The intuition here is that since the weights add up to less than one, the platform would be under-utilized if real time ticked at the same speed as virtual time; we speed up virtual time instead of allowing idleness. The last

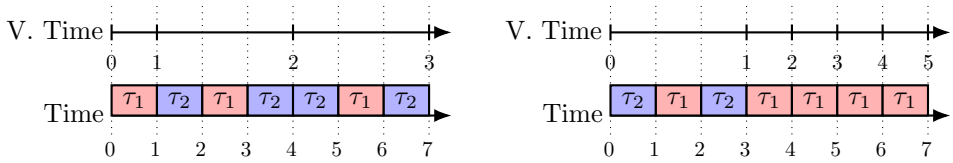

Figure 2: *In the schedule on the left, τ_1 releases its first request immediately at $t = 0$ and τ_2 releases its first at $t = 1$. Each issue subsequent requests as soon as the previous request is fulfilled. Using (8), we see that $V(1) = V(0) + \frac{1}{1}(1-0) = 1$; virtual time ticks at the same rate as real time in $[0, 1)$. After τ_2 joins the competition at $t = 1$, we calculate $V(2)$ using (8) as $V(2) = V(1) + \frac{1}{1+2}(2-1) = 1 + \frac{1}{3}$, since the weights are constant in $[1, 2)$. Virtual time slows to tick at $\frac{1}{3}$ the rate of real time and continutes on that way, just as in Figure 1.*
In the schedule on the right, each τ_i issues its first request at $t = 0$. Client τ_1 issues subsequent requests as soon as the previous request is fulfilled and client τ_2 leaves the competition as soon as its first request is fulfilled. The slope of virtual time with respect to real time is the same as in the schedule on the left when the same clients are active. Section 4 contains further discussion of dynamic systems, including the computation of virtual time in the case that τ_2 leaves the competition before its request is fulfilled.

case is if $\frac{1}{\sum_j w_j} < 1$, virtual time ticks slower than real time. The intuition here is the opposite as the previous case; we must slow virtual time down in order to prevent the system from being over-utilized.

Virtual time is closely related to ideal allocation; we have the following useful relation:

$$S_i(t_1, t_2) = (V(t_2) - V(t_1))w_i\,, \tag{9}$$

assuming w_i is constant in $[t_1, t_2)$. If this is not the case, one can compute $S_i(t_1, t_2)$ by a finite piecewise expression of terms of the form in (9).

Proof. We prove Eq. (9) follows from our definitions.

In this proof, we introduce the following notation that will be used throughout the paper.

$$\begin{gathered} A \\ =\{B\} \\ C \\ \Rightarrow\{D\} \\ E \end{gathered}$$

This means that A is equal to C by B and that C implies E by D.
For $[t_1, t_2)$ where the weights in the system are constant, (6) gives

$$
\begin{aligned}
& V(t_2) = V(t_1) + \frac{1}{\sum_{j \in A(t_1)} w_j}(t_2 - t_1) \\
=& \{\text{rearrange}\} \\
& \frac{1}{\sum_{j \in A(t_1)} w_j}(t_2 - t_1) = V(t_2) - V(t_1) \\
=& \{\text{multiply by } w_i\} \\
& \frac{w_i}{\sum_{j \in A(t_1)} w_j}(t_2 - t_1) = w_i(V(t_2) - V(t_1)) \\
=& \{(1)\} \\
& f_i(t_1)(t_2 - t_1) = w_i(V(t_2) - V(t_1)) \\
=& \{\text{weights constant in } [\mathrm{t}_1, \mathrm{t}_2)\} \\
& \int_{t_1}^{t_2} f_i(t)\, dt = w_i(V(t_2) - V(t_1)) \\
=& \{(2)\} \\
& S_i(t_1, t_2) = w_i(V(t_2) - V(t_1))\,,
\end{aligned}
$$

for all $[t_1, t_2)$ where w_i constant. □

We find that the ideal allocation of τ_i, $S_i(t_1, t_2)$, can be expressed as the virtual time elapsed in $[t_1, t_2)$ scaled by w_i. This is logical considering client τ_i executes for w_i real time units in each virtual time unit. Virtual time here functions as an alternate way to reason about share of the system over time in calculating ideal allocation.

We now introduce the EEVDF algorithm to schedule requests *in* virtual time with the goal of approximating the fluid system.

2.3 The EEVDF Algorithm

We begin this subsection by stating the *Earliest Eligible Virtual Deadline First (EEVDF)* algorithm. We then define a client's *eligible time* and *virtual deadline*, which are parameters to the algorithm.

Definition 1. *EEVDF Algorithm: A new quantum is allocated to the client which has the eligible request with the earliest virtual deadline.*

In a system τ, client τ_i may issue any number of requests r_i^0, r_i^1, ..., r_i^k. Note that different requests by the same client need not be the same length. After each request is fulfilled, τ_i either issues a new request or becomes passive. Again, a client may have at most one active request at one time.

We now introduce the notions of *eligible time* and *deadline*. Each request r_i^k has an associated eligible time e_i^k and deadline d_i^k in the real time domain.

Since an active client is one that has an active request, each active client τ_i always has an associated e_i and d_i. A client's eligible time is the time at which it is eligible to be allocated service time by the scheduler and its deadline is the time by which it must have its current request fulfilled in order to approximate the fluid system.

EEVDF is a scheduling algorithm for dynamic systems, systems in which clients may leave or join at any point. We will formalize the notion of a dynamic system in Section 4, but now simply state that the fluid system is defined as the proportional uniform progress of all clients over time, and that the leaving and joining of clients will naturally change the rate of each client's uniform progress. Since the goal of the EEVDF algorithm algorithm is to approximate this fluid system, we find that we must recompute each client's e_i and d_i in the real time domain after each change in the dynamic system. In order to avoid this prohibitively expensive computation, we instead compute each client τ_i's *virtual eligible time* and *virtual deadline* and make scheduling decisions in the virtual time domain. To understand this mechanism, we present definitions of e_i^k, d_i^k, $V(e_i^k)$, and $V(d_i^k)$ for a request r_i^k.

To help motivate the following definitions, we restate that the goal of the EEVDF is to imitate the ideal schedule as closely as possible by *minimizing lag*. After the discussion on virtual time, we make a clarifying note that the units of each term in (6) are *real time* units; $\text{lag}_i(t) = 3$ means that client τ_i has received 3 real time units more allocation in the ideal system than the actual system.

Suppose client τ_i releases a request r_i^k at time t_i^k. In the deriving the following two equations, for notational brevity, we let $x = x_i^k$ for $x = \{r, e, d\}$.

We define e_i^k as the point in time where the time allocated to τ_i in the actual system before issuing r_i^k is equal to the time allocated to τ_i in the fluid system, or $s(t_i^0, e_i^k) = S(t_i^0, t_i^k)$. In other words, e_i^k should be the time in the real time domain that τ_i is scheduled in the corresponding fluid system. Using (9), we obtain the virtual eligible time

$$S_i(t_i^0, e) = s_i(t_i^0, t_i^k) \tag{10}$$

$$(V(e) - V(t_i^0))w_i = s_i(t_i^0, t_i^k)$$

$$V(e) = V(t_i^0) + \frac{s_i(t_i^0, t_i^k)}{w_i}\,. \tag{11}$$

A client's eligible time is tied to its lag. Specifically, a client should be scheduled in order to minimize its lag. Lemma 1 of the Appendix states that the lag of an active client strictly increases over time when it is not scheduled and decreases over time when it is. It follows that a client should be eligible when it has non-negative lag and ineligible when it has negative lag, behavior that is achieved through this definition of eligible time. In fact, a client is only eligible to be scheduled when its lag is non-negative. To see why, consider the two cases for a request r_i^k released at time t_i^k.

Case 1. Request r_i^k is eligible to be scheduled as soon as it is released, or $e_i^k \leq t_i^k$. Then from (10), we have $S_i(t_i^0, t_i^k) \geq s_i(t_0^i, t_i^k)$, or that $\text{lag}_i(t_i^k) \geq 0$.

This is the case where τ_i at t_i^k is behind its allocation in the ideal system; it is "owed time" and must catch up, thus it is eligible to be scheduled immediately.

Case 2. Request r_i^k is not eligible to be scheduled as soon as it is released, or $e_i^k > t_i^k$. In this case, we have $S_i(t_0^k, t_i^k) < s_i(t_0^k, t_i^k)$ and $\text{lag}_i(t_i^k) < 0$, resulting from the fact that $S_i(e_i^k, t_i^k) = \int_{e_i^k}^{t_i^k} f_i(t)dt = -\int_{t_i^k}^{e_i^k} f_i(t)dt < 0$ by (3),(2), and $e_i^k > t_i^k$.

We now define our deadline to represent the time that the fluid system would complete request r_i^k of length r, having started at the eligible time and having made uniform progress. We choose the deadline d_i^k such that $S_i(e, d) = r$ and, using (9), we obtain the virtual deadline

$$S_i(e, d) = r \tag{12}$$
$$(V(d) - V(e))w_i = r$$
$$V(d) = V(e) + \frac{r}{w_i}\,. \tag{13}$$

An important observation here is that $V(e_i)$ and $V(d_i)$ depend only on the parameters of τ_i and do not vary as clients leave or join the competition. There is therefore no need to recompute them as dynamic changes happen in the system. However, because we cannot know the future weights of the clients in the system, we cannot compute the future rate at which virtual time will tick against real time in the future. Thus, we cannot provide the real-time value d_i corresponding to $V(d_i)$ at a real time $t < d_i$. The same is true for e_i. Thus, we make all scheduling decisions in the virtual-time domain and map virtual time to real time at different rates as clients leave and join. In this way, the virtual-time domain allows us to avoid having to recompute eligible times and deadlines after changes to the system while still approximating the ideal system.

Before providing an example of a task set scheduled under EEVDF, we provide a set of equations that will be useful in reasoning about the example.

First, we make the simplifying assumption that a client τ_i that has made a request of length $|r_i^k|$ will use all of its requested time *i.e.,* that it will not finish early or leave the competition before its request is fulfilled. We will relax this assumption in Section 4. Further, we assume that the client issues a new request after each of its requests is fulfilled. Under these assumptions, by (11) and (13), we find for a client τ_i

$$V(e^1) = V(t_i^0) \tag{14}$$
$$V(d^k) = V(e^k) + \frac{|r^k|}{w_i} \tag{15}$$
$$V(e^{(k+1)}) = V(d^k)\,, \tag{16}$$

where r_i^1 is τ_i's first request.

Now that we have defined EEVDF and seen an example in Figure 3, we make some general statements about the algorithm.

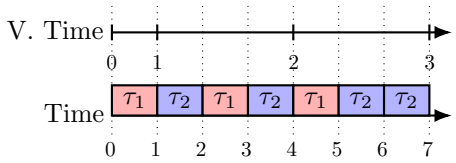

Figure 3: *An example EEVDF schedule of system* $\tau = \{\tau_1, \tau_2\}$ *where* $w_1 = 1$ *and* $w_2 = 2$*, the same system in Figure 1, Figure 2. We take* $q = 1$*. We assume all clients issue requests for time equal to their weight, use all of their requested time and issue new requests as soon as their previous one is fulfilled. Client* τ_1 *releases its first request at time* 0 *and* τ_2 *at time* 1*. Note that the mapping between virtual and real time is detailed in Figure 2. The only active client in the system at time* 0 *is* τ_1*; it is eligible by (14) and has a deadline* $V(d_1^1) = 0 + \frac{1}{1} = 1$ *by (15). It is scheduled for one quantum at time* 0 *and finishes at its virtual deadline. Now we look at time* 1*. We have* τ_1 *is eligible by (16); it has eligible time* $V(e_1^2) = V(d_1^1) = 1$ *and deadline* $V(d_1^2) = 1 + \frac{1}{1} = 2$ *from (15). It is no longer the only client in the system; at time* 1*,* τ_2 *becomes active and eligible by (14). Its virtual deadline is virtual time* 2 *by (15):* $V(d_2^1) = 1 + \frac{2}{2} = 2$*. Both* τ_1 *and* τ_2 *have the same virtual deadline; we break the tie arbitrarily in favor of* τ_2 *both at time* $t = 1$ *and* $t = 2$*. At time* $t = 4$*, we break the tie in favor of* τ_1 *to illustrate that lag remains bounded. Using (14), (15), (16), we obtain the remaining allocations in the schedule.*

First, the EEVDF algorithm is a *work-conserving* algorithm, which means that as long as there exist active requests in the system, the CPU will never idle. This property is stated as Lemma 4 and proved in the Appendix.

Second, we state that at all times t, all clients have bounded lag under the EEVDF algorithm. Specifically, given a system with a time quantum of size q in which clients make requests of length less than or equal to q, the lag of any client at any time is asymptotically bounded by $-q$ and q. This property is discussed in Section 7 and proved in the Appendix. This bound represents a bounded error between the actual system under EEVDF and the fluid system in which each client makes uniform *fair* progress. Then, we understand EEVDF as an algorithm in which fairness is first-class concern. This is to say that we begin by making some statement about fairness: the parameters to our algorithm are defined with respect to the ideal, fair system, and derive some statement about timeliness: a request will always be fulfilled within a quantum q of its deadline.

Lastly, we discuss the parameters by which EEVDF "prioritizes" work: the length of a client's request and its weight. Client τ_i's virtual deadline, by (13), is explicitly a function of these two parameters, w_i and request size r_i. A client with a smaller weight will have a longer deadline, and will thus be generally less prioritized than clients with larger weights. A client that makes longer request will also have longer deadlines, and will consequently be less prioritized than those who make shorter requests. Clients with larger weights and who make

shorter requests are conversely more prioritized by being scheduled generally earlier under EEVDF. Then, in any implementation of the EEVDF algorithm, we find the natural tradeoff between latency and throughput; shorter requests guarantee better allocation accuracy while longer ones decrease system overhead.

3 Related Work

This paper, as previously stated, is intended as an extension to the work on EEVDF by Stoica and Abdel-Wahab in 1995 [1] and developed further by Stoica et al. [2] in 1996. We now give a very brief survey of related fair scheduling algorithms.

The Pfair algorithm, introduced by Baruah et al. in 1995 [3], like the EEVDF algorithm, enforces a uniform rate of execution of work, elevating fairness to a first-class concept. This algorithm, as well, assigns tasks a weight, allocates processor time in discreet quanta, and relies on the concept of lag or error with respect to an ideal system. Pfair works by mandating that a task's lag at any time t must be bounded by a constant; deadline guarantees are derived from this definition. We contrast this with EEVDF, in which the algorithm is defined in relation to deadlines and lag bounds are derived. Last, we note that the Pfair algorithm, unlike EEVDF, is explicitly a multiprocessor scheduling algorithm.

PD^2, introduced by Srinivasan and Anderson in 2002 [4], is another fair multiprocessor scheduling algorithm that utilizes the concept of lag with respect to an ideal system.

Another extremely influential concept in fair scheduling is the lottery scheduler, introduced by Waldspurger and Weihl in 1994 [5], which uses probabilistic mechanisms to ensure fairness. The lottery scheduler assigns each process in the system a weight. When making a scheduling decision, each process has a chance of being chosen equal to its share of the total weight in the system. In this sense, the weight can be thought of as a number of "lottery tickets" where a process with a higher weight (more lottery tickets) has a higher chance of being chosen to run on the processor.

Lastly, we reference the *Completely Fair Scheduler* (CFS), the previous default fair scheduler used by the Linux kernel [6]. The CFS uses the concept of virtual time to allocate proportional CPU time, but does not strictly honor deadlines and relies on heuristics to handle latency-sensitive work. The CFS uses a red-black tree to keep a run-queue of clients, ordered by `vruntime`. The value of `vruntime` is given by $\frac{\texttt{runtime}}{\texttt{weight}}$ and the client with the smallest `vruntime` at any point has the highest priority. The benefits of EEVDF over the CFS are further discussed in Section 5.

4 EEVDF in Dynamic Systems

We define a dynamic system as one in which clients can leave, join, or change their weights at any point in time. A client changing its weight is not considered a third operation, but as a client leaving the system and then rejoining with a different weight. We precisely define a client τ_i leaving the system as changing its weight to $w_i = 0$. A client leaving the system is the same as a client becoming passive. We will call a client leaving, joining, or changing its weights a *dynamic operation*.

The first important observation in considering EEVDF for dynamic systems is that supporting dynamic operations in the fluid system is trivial. Because all clients make uniform progress in the fluid system, dynamic operations cause only a change in the rate of progress for each client.

The second important observation is that EEVDF as we have described it so far supports dynamic operations for clients at points where their lag is zero. If a client τ_k leaves or joins the system at a point t such that $\text{lag}_k(t) = 0$, the mapping between real and virtual time will be adjusted according to (8) to reflect the new weights in the system and scheduling decisions will continue to be made by the EEVDF policy, as seen in Figure 2. If, however, clients may execute dynamic operations at points where their lag is nonzero, the description of the EEVDF algorithm presented thus far is insufficient.

Before presenting the extension of the algorithm so far to the dynamic case, we motivate the work with the question: Why is it important to allow dynamic operations at non-zero lag points? The answer is that allowing clients to leave/join at any time and to not use all of their requested time is vital for responsive systems. For example, we do not want to delay the joining of a latency-sensitive client or force the system to idle if a client with non-zero lag requests to leave. EEVDF is designed to be fair algorithm for use in a wide range of general computing contexts, so responsiveness is a high-class concern. Furthermore, the ability for a client to not use all of its requested time and not be punished allows request time to become a latency parameter, allowing clients to more easily influence their treatment by the scheduler.

We give an overview of the subsections in this section. In Section 4.1, we present the complete EEVDF policy for dynamic systems. In Section 4.2, examples of and discussion about the policy. In Section 4.3 a formal derivation of the policy. In Section 4.4, a survey of how the policy is applied in related work and in the Linux EEVDF implementation.

4.1 EEVDF Policy for Dynamic Systems

The following two rules comprise the policy of EEVDF for dynamic systems. The notation t^+ refers to the time just after t, with $t^+ \to t$.

R1. If a client τ_k *leaves* the system at time t, the value of virtual time is updated:

$$V(t^+) = V(t) + \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j}.$$

R2. If a client τ_k *joins* the system at time t, the value of virtual time is updated:

$$V(t^+) = V(t) - \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j} .$$

When a client leaves or joins the system with non-zero lag, these equations reduce to $V(t^+) = V(t)$, or no update. A client changing its weight is handled as an application of **R1** followed by **R2**. If multiple dynamic operations happen at the same time, they are simply handled as repeated applications of the appropriate rules.

For **R1**, in the case that the leaving client, τ_k, was the only client in the system, we let $V(t^+) = V(t)$. Because the slope of virtual time is defined to be 0 at points with no active clients, the value of virtual time will remain fixed until the next client becomes active. This point is of primarily theoretical interest and is included for completeness; there is almost never a case where an actual system is without a single active process.

We make the observation that a client with negative lag leaving has the same effect as a client with positive lag joining.

When a client leaves, joins, or changes its weight, the value of virtual time is updated according to rules **R1** and **R2** and the slope of virtual time is updated according to (8). We note that this is a generalization of the case where a client leaves/joins/changes weight with zero lag: in that case, rules **R1** and **R2** give $V(t^+) = V(t)$ and in both cases, the slope of virtual time is updated according to (8).

At this point, we have presented every way virtual time can be modified. So, we can now see that in an implementation of EEVDF, all that needs to be done to track virtual time is to keep a variable that is updated after each event—either scaled by the inverse of weights in the system between scheduling decisions, or updated discretely after dynamic operations.

4.2 Examples and Discussion

As we will see in the following section, rules **R1** and **R2** are a consequence of assuming the following invariant must hold for any implementation of EEVDF: at any time t, the sum of the lags of all active clients is zero, *i.e.,*

$$\sum_{i \in A(t)} \text{lag}_i(t) = 0 . \tag{17}$$

This invariant is important because it means that the lag of a client accurately describes its progress in relation to its progress in the ideal system. To see this, note that (17) is equivalently expressed as

$$\sum_{i \in A(t)} S_i(t_i^0, t) = \sum_{i \in A(t)} s_i(t_i^0, t) .$$

So, if (17) does not hold, the service time provided to clients in the ideal system is not the same as the service time provided to clients in the actual

system, and the systems are no longer comparable; the lag of a client no longer refers to something meaningful.

Practically, this invariant means that if a client has negative lag at a point, the sum of the lags of the remaining clients at that point must be positive. In other words, if a client has received more service time than it was entitled to, the remaining clients must have collectively received less service time than they were entitled to. The same is true for a client that has a positive lag at a point: this client has received less service time than it was entitled to and the other clients have collectively received more service time than they were entitled to. Said another way, a loss for one client is a gain for all the others and a gain for one client is a loss for all the others.

So, if a client leaves or joins with non-zero lag and we do nothing to adjust the system, our invariant is violated. By updating virtual time with **R1** and **R2**, we maintain the invariant by proportionally redistributing the lag of the leaving/joining client to the other clients in the system.

To see this explicitly, consider the following two equations.

Case 1. Client τ_k leaves at time t.

$$\text{lag}_i(t^+) = \text{lag}_i(t) + w_i \frac{\text{lag}_k(t)}{\sum_{j \neq k}}, i \neq k\,.$$

Case 2. Client τ_k joins at time t.

$$\text{lag}_i(t^+) = \text{lag}_i(t) - w_i \frac{\text{lag}_k(t)}{\sum_{j \neq k}}, i \neq k\,.$$

Proof.

Case 1. Client τ_k leaves at time t. Consider $\tau_i \neq \tau_k$ From (6),

$$\begin{aligned}
\text{lag}_i(t^+) &= S_i(t_i^0, t^+) - s_i(t_i^0, t^+) \\
&= \{(9)\} \\
\text{lag}_i(t^+) &= w_i(V(t^+) - V(t_i^0)) - s_i(t_i^0, t^+) \\
&= \{\mathbf{R1}\} \\
\text{lag}_i(t^+) &= w_i(V(t) + \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j} - V(t_i^0)) - s_i(t_i^0, t^+) \\
&= \{\text{rearrange}\} \\
\text{lag}_i(t^+) &= w_i(V(t) - V(t_i^0)) - s_i(t_i^0, t^+) + w_i \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j} \\
&= \{(9)\} \\
\text{lag}_i(t^+) &= S_i(t_i^0, t) - s_i(t_i^0, t^+) + w_i \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j} \\
&= \{t^+ \to t\}
\end{aligned}$$

$$\begin{aligned}\text{lag}_i(t^+) &= S_i(t_i^0, t) - s_i(t_i^0, t) + w_i \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j} \\ &= \{(6)\} \\ \text{lag}_i(t^+) &= \text{lag}_i(t) + w_i \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j} \,.\end{aligned}$$

Case 2. Client τ_k joins at time t. Using **R2** instead of **R1** in the previous derivation, we have

$$\text{lag}_i(t^+) = \text{lag}_i(t) - w_i \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j} \,, i \neq k.$$

□

In both equations, we see that the lag of the client τ_k is proportionally distributed among the other clients in the system in the case of a dynamic operation.

We now present two examples of **R1**. In each of the following examples, two schedules are presented to illustrate the policy. The one on the top assumes virtual time is not not adjusted using **R1** while the one on the bottom assumes that it is; the schedule on the bottom is the correct schedule produced by EEVDF. In each of the schedules, we have three clients, $w_1 = w_2 = w_3 = 1$. Each client releases its first request at $t = 0$.

In Figure 5, we see an example of virtual time being updated with a smaller value. To avoid confusion in these cases, we must remember that virtual time is a mathematical tool that helps us reason about performance of the actual system with respect to the ideal system. Virtual time allows us to make statements about deadlines of clients relative to each other in an tractable time as dynamic events take place in the system. There is no guarantee that virtual time increases as real time increases. Real time is strictly increasing and once a client has been scheduled in real time, that allocation will never be undone. Virtual time, then, can be thought of as a parameter that determines which requests are eligible at any given real time.

Lastly, we state that by updating the value of virtual time by **R1** and **R2**, we ensure bounded lag for any client in the system at any time. This property is proved in [1].

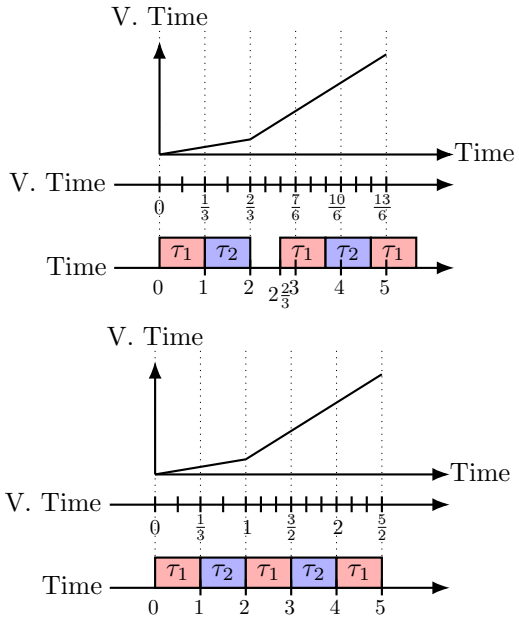

Figure 4: ***Client*** τ_3 ***leaves with positive lag at*** $t = 2$. *At* $t = 2$, *each client has executed in the fluid schedule for* $\frac{2}{3}$ *time units, while only* τ_1 *and* τ_2 *have been allocated time on the processor. Then,* $\text{lag}_3(2) = \frac{2}{3}$, $\text{lag}_1(2) = \text{lag}_2(2) = -\frac{1}{3}$. *In the top schedule we see by (9) that the lags of* τ_1, τ_2 *are negative until* $V(3) = 2\frac{2}{3}$ *if we do not adjust virtual time after* τ_3 *leaves, meaning we will have no eligible clients in the system until that point, and the system is idle for* $[2, 2\frac{2}{3})$. *In the bottom schedule, we avoid this idle time by updating the virtual time at* $t = 2$ *using* ***R1****:* $V(2+) = V(2) + \frac{\text{lag}_3(2)}{w_1+w_2} = \frac{2}{3} + \frac{1}{3} = 1$. *Now,* τ_1, τ_2 *are eligible to be scheduled at* $V(2)$. *In the positive lag leaving case, we can think of* ***R1*** *as "skipping ahead" in virtual time to avoid the processor idling.*

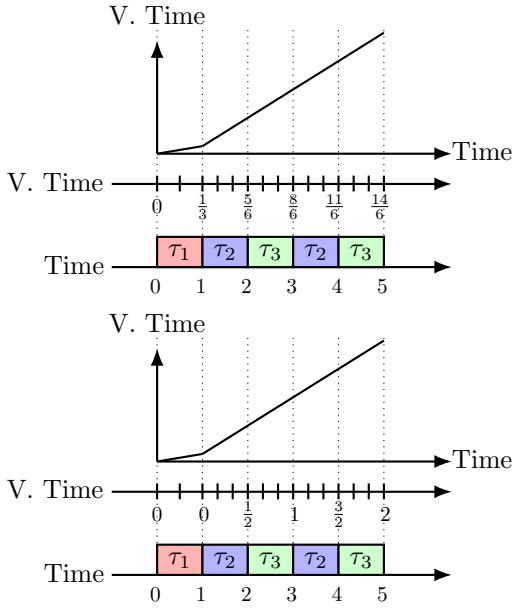

Figure 5: ***Client*** τ_1 ***leaves with negative lag just before*** $t = 1$***.*** *Then by (6), we have* $\text{lag}_2(1) = \text{lag}_3(1) = \frac{1}{3}$ *and* $\text{lag}_1(1) = -\frac{2}{3}$*. Since* τ_1 *leaves the system,* $\sum_{i \in A(1^+)} \text{lag}_i(1^+) = \frac{1}{3} + \frac{1}{3}$*. This violates our lag invariant, (17). In fact, if we do not update virtual time according to* ***R1****, this violation will not disappear. To see this, consider* $\text{lag}_2(3) = \text{lag}_3(3) = (\frac{8}{6} - 0) - 1 = \frac{1}{3}$*, by (6) and (9). In this schedule, we find* $\text{lag}_2(1+2k) = \text{lag}_3(1+2k) = ((\frac{1}{3} + k) - 0) - k = \frac{1}{3}$ *for all* $k \in \mathbb{Z}$*. Then,* τ_2 *and* τ_3 *carry around their lag of* $\frac{1}{3}$ *forever. Then, the lag of the remaining clients can become arbitrarily large as an arbitrary number of clients with negative lag leave the system. However, by updating virtual time using* ***R1****, as in the schedule on the bottom,* $V(1^+) = V(1) + \frac{\text{lag}_1(1)}{w_1 + w_2} = \frac{1}{3} + \frac{\frac{-2}{3}}{2} = 0$*, we distribute the negative lag of* τ_1 *fairly among the remaining clients and maintain lag invariant (17). See:* $\text{lag}_2(1) = \text{lag}_3(1) = (V(1) - V(0))w_2 = (0 - 0)1 = 0$ *from (6), (9).*

4.3 Derivation of Rules for Dynamic Operations

In this subsection, we derive rules **R1** and **R2** from the lag invariant (17).

We first derive **R1**, the rule for a client leaving the system. Suppose τ_k leaves at time t. Let t^+ be the moment of time right after the client leaves; $t^+ \to t$. We must show that $V(t) = V(t^+) + \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j}$.

First, we note that in this case we have

$$\sum_{i \in A(t^+)} \text{lag}_i(t) = \sum_{i \in A(t)} \text{lag}_i(t) - \text{lag}_k(t) = -\text{lag}_k(t)\,. \tag{18}$$

Expanding the definition of lag, (6), we find

$$\begin{aligned}
&\text{lag}_i(t^+) = S_i(t_i^0, t) + S_i(t, t^+) - s_i(t_i^0, t) - s_i(t, t^+) \\
=&\{(6)\} \\
&\text{lag}_i(t^+) = \text{lag}_i(t) + S_i(t, t^+) - s_i(t, t^+) \\
=&\{(t^+ \to t) \Rightarrow s_i(t, t^+) = 0\} \\
&\text{lag}_i(t^+) = \text{lag}_i(t) + S_i(t, t^+) \\
=&\{(9)\} \\
&\text{lag}_i(t^+) = \text{lag}_i(t) + (V(t^+) - V(t))w_i \\
\Rightarrow&\{\text{algebra}\} \\
&\sum_{i \in A(t^+)} \text{lag}_i(t^+) = \sum_{i \in A(t^+)} (\text{lag}_i(t) + (V(t^+) - V(t))w_i) \\
\Rightarrow&\{(17)\} \\
&\sum_{i \in A(t^+)} (\text{lag}_i(t) + (V(t^+) - V(t))w_i) = 0 \\
=&\{\text{algebra}\} \\
&[\sum_{i \in A(t^+)} \text{lag}_i(t)] + (V(t^+) - V(t)) \sum_{i \in A(t^+)} w_i = 0 \\
=&\{(18)\} \\
&-\text{lag}_k(t) + (V(t^+) - V(t)) \sum_{i \in A(t^+)} w_i = 0 \\
=&\{\text{algebra}\} \\
&V(t) = V(t^+) + \frac{\text{lag}_k(t)}{\sum_{i \in A(t^+)} w_i} \\
=&\{\sum_{i \in A(t^+)} w_i = \sum_{j \neq k} w_j\} \\
&V(t) = V(t^+) + \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j}\,,
\end{aligned}$$

as intended.

Now we derive **R2**, the rule for a client joining the system. Suppose τ_k joins at time t. Let t^+ be the moment of time right after the client joins; $t^+ \to t$. We must show that $V(t) = V(t^+) - \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j}$.

In this case, we have

$$\sum_{i \in A(t^+)} \text{lag}_i(t) = \sum_{i \in A(t)} \text{lag}_i(t) + \text{lag}_k(t) = \text{lag}_k(t)\,. \tag{19}$$

We note that the derivation proceeds the same as the previous, save for using (19) in place of (18).

Again we expand the definition of lag, (6), and find

$$\begin{aligned}
&\text{lag}_i(t^+) = S_i(t_i^0, t) + S_i(t, t^+) - s_i(t_i^0, t) - s_i(t, t^+) \\
=&\{(6)\} \\
&\text{lag}_i(t^+) = \text{lag}_i(t) + S_i(t, t^+) - s_i(t, t^+) \\
=&\{(t^+ \to t) \Rightarrow s_i(t, t^+) = 0\} \\
&\text{lag}_i(t^+) = \text{lag}_i(t) + S_i(t, t^+) \\
=&\{(9)\} \\
&\text{lag}_i(t^+) = \text{lag}_i(t) + (V(t^+) - V(t))w_i \\
\Rightarrow&\{\text{algebra}\} \\
&\sum_{i \in A(t^+)} \text{lag}_i(t^+) = \sum_{i \in A(t^+)} (\text{lag}_i(t) + (V(t^+) - V(t))w_i) \\
\Rightarrow&\{(17)\} \\
&\sum_{i \in A(t^+)} (\text{lag}_i(t) + (V(t^+) - V(t))w_i) = 0 \\
=&\{\text{algebra}\} \\
&[\sum_{i \in A(t^+)} \text{lag}_i(t)] + (V(t^+) - V(t)) \sum_{i \in A(t^+)} w_i = 0 \\
=&\{(19)\} \\
&\text{lag}_k(t) + (V(t^+) - V(t)) \sum_{i \in A(t^+)} w_i = 0 \\
=&\{\text{algebra}\} \\
&V(t) = V(t^+) - \frac{\text{lag}_k(t)}{\sum_{i \in A(t^+)} w_i} \\
=&\{\sum_{i \in A(t^+)} w_i = \sum_{j \neq k} w_j\} \\
&V(t) = V(t^+) - \frac{\text{lag}_k(t)}{\sum_{j \neq k} w_j}\,,
\end{aligned}$$

as intended.

We have proven that (17) implies **R1** and **R2**. In Lemma 2 of [1], Stoica and Abdel-Wahab prove that **R1** and **R2** (taken together) imply (17). So, these are equivalent conditions.

4.4 Implementation Strategies

So far in Section 4, we have described the most general implementation of EEVDF for dynamic systems. In a real-life implementation, however, there are further considerations to be made. For example, should a client carry negative lag indefinitely, *i.e.,* be punished for execution that happened an arbitrary length of time in the past? Similarly, should a client carry positive lag indefinitely, *i.e.,* be owed service time from missed execution time an arbitrary length of time in the past?

In this section, we provide a survey of the strategies employed by [2], [1], and by the current EEVDF Linux implementation for dealing with dynamic EEVDF systems, specifically their policies for clients that leave/join with non-negative lag.

Implementation in [1]: This is the original work proposing EEVDF, by Stoica and Abdel-Wahab. Three implementations are proposed in their Sec. 5, titled Algorithm Implementation.

1. Use rules **R1** and **R2** whenever a client leaves or joins, respectively.
2. Apply **R1** when a client leaves, but to forget the client's lag upon rejoining, *i.e.,* any client that joins the competition has zero lag.
3. Allow a client to execute dynamic operations only when its lag is zero.

Implementation in [2]: This work by Stoica et al. proposes a single theoretical implementation of EEVDF.

In it, clients that leave with *positive lag* cause an update of virtual time according to **R1** and, just as in implementation two of [1], all lag is forgotten upon rejoining so clients always join with zero lag.

Clients that leave with *negative lag*, instead of actually leaving the system when they request to, are delayed (without being allocated time) to leave at the point that their lag becomes zero. By Lemma 1, the lag of a client is strictly increasing as long as that client is not scheduled, so this point always exists. In the paper, this is accomplished by having the leaving client issue a dummy request of length zero that will be serviced at the next point that the client is eligible, which by definition (10) is the next point at which the client's lag is zero. Through this mechanism, this implementation avoids updating virtual time in the negative lag leaving case and eliminates the negative lag joining case.

In this implementation, all clients join the system with zero lag and virtual is time strictly increasing.

Linux Implementation: In the Linux kernel, [7], clients may leave or join the system at any point in time, regardless of lag value. A client that leaves with

a positive lag will retain its positive lag indefinitely and rejoin the competition with the same positive lag it had when it left. A client that leaves with negative lag will have its negative lag decayed at the rate of virtual time until it reaches zero, *i.e.,* its lag will increase at the rate of its weighted uniform progress. In this manor, negative lag is forgiven, but not so quickly that a malicious client could exploit the negative lag forgiveness mechanism by leaving and rejoining at negative lag points to win more than its fair share of processor time.

Since the concept of virtual time from the paper does not map directly to the Linux implementation, we cannot say that rules **R1** and **R2** are directly applied. The next section, Section 5, provides a more detailed discussion on the dealing of dynamic systems in the Linux implementation.

5 EEVDF in Linux

The Linux kernel documentation ([7], [8], [9]) explicitly cites the 1995 paper by Stoica and Abdel-Wahab, [1], in which the algorithm was first introduced, as the theoretical basis for its EEVDF implementation. The EEVDF algorithm was merged as an option for the 6.6 Linux kernel, released October 29, 2023, replacing the "Completely Fair Scheduler" (CFS) as the default scheduler for the fair scheduling classes (see Figure 6) [10], [7]. The CFS was implemented by Ingo Molar and merged in Linux 2.6.23 (October 9, 2007) [11]. In the `sched/fair.c` source code [9], where the bulk of the EEVDF-based scheduler is implemented, the term CFS is still widely used to refer to the implementing structures and logic. This is because rather than implement an entirely new EEVDF scheduler from scratch, much of the old CFS code was kept as just the relevant bits of logic were replaced with EEVDF.

The change from the CFS to EEVDF was made in large part to address difficulties in managing processes with varying latency requirements. Where the CFS relied on a collection of heuristics, the EEVDF policy is naturally suited to handle these types of workloads. Under EEVDF, clients which issue longer requests will have longer virtual deadlines and will thus be effectively de-prioritized under EEVDF compared to clients which request shorter time-slices, which are often requested by processes with stricter latency requirements, *e.g.*, graphical or network tasks sleeping frequently on I/O requests.

A schedulable entitiy in Linux is often referred to as a *task*, so that is the name we will adopt for this section in place of *client*.

5.1 Scheduling Classes

Each task in Linux belongs to a scheduling class. Each scheduling class is associated with a scheduling algorithm. The Linux scheduling classes are given in Figure 6 in order of priority, with the highest priority at the top of the table.

As described in [12], the stop class has the highest priority and is only used by the kernel for maintenance tasks. The deadline class is used for real-time work and uses the Earliest Deadline First (EDF) algorithm. The FIFO and RR

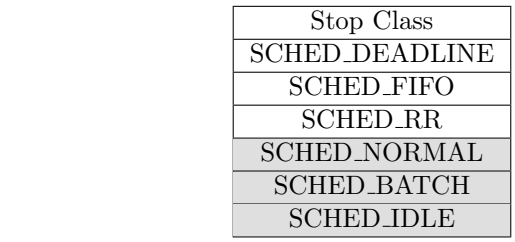

Figure 6: Linux Scheduling Classes

classes are for real-time work and use the First-in-First-Out and Round Robin algorithms, respectively. The normal, batch, and idle classes are referred to as *fair* scheduling classes and use the EEVDF algorithm. The SCED_NORMAL class is referred to as SCED_OTHER in the POSIX standard; is it used for non-real-time, interactive work. The batch class is used for non-real-time, non-interactive work. The idle class has the lowest priority; tasks in this class will execute only when no other work exists in the other classes.

The kernel maintains a run-queue for each scheduling class on each CPU. Each run-queue is ordered by the policy of its scheduling_class, *i.e.,* EDF for SCED_DEADLINE or EEVDF for SCHED_NORMAL. When a scheduling decision is to be made, the core scheduler iterates over the scheduling classes in order of priority, finds the highest priority scheduling class with a runnable task, and schedules the task at the head of its run-queue. A task may be preempted if a task in a higher priority scheduling class becomes runnable during its execution [12].

5.2 Linux Implementation of EEVDF

A task, or client in the language of the paper, is represented in Linux as a `sched_entity` struct. The Linux Implementation of EEVDF uses a red-black tree to maintain a run-queue of `sched_entity` ordered by virtual deadline. Enqueue, dequeue, and search operations in the red-black tree are O(log n). When choosing a task to schedule, the EEVDF picks the leftmost *eligible* node, representing the eligible task with the earliest deadline. The Linux implementation of EEVDF is preemptable; tasks are preempted by tasks with earlier virtual deadlines.

Each `sched_entity` indirectly tracks its runtime as `vruntime`, or v_i, found by scaling its runtime by the inverse of its weight. A task's `vruntime` is updated after each time the task is allocated time on the CPU using

$$v_i(t_2) = v_i(t_1) + \frac{s_i(t_1, t_2)}{w_i},$$

for $t_1 < t_2$. In the previous Linux fair scheduler, the CFS, the run-queue was sorted by least `vruntime` [12].

Virtual time in the Linux implementation is tracked using a variable called V. This *approximation* of the paper's virtual time is maintained for each instance of EEVDF — that is, each for each scheduling class on each CPU. This variable represents the weighted average of all active entity virtual runtimes; the value of V at any time t is given by

$$V(t) = \frac{\sum_{\tau_j \in A(t)} (w_j * v_j(t))}{\sum_{\tau_j \in A(t)} w_j} .$$

This V gives the value of virtual time at any point in the system; it ticks along in inverse proportion to the sum of weights in the system, changing discreetly at points that entities with non-zero lag leave or join. This is because since V is just the average of all virtual runtimes of entities currently in the system, it can be pulled down or pushed up when entities with `vruntime` above or below the average leave/join [13].

The lag for each client is mapped to the concept of `vlag` in the Linux implementation, given by

$$\text{vlag}_i(t) = w_i(V(t) - v_i(t)) .$$

Among other fields, the `sched_entity` struct contains fields for `deadline`, `slice`, `vruntime`, and `vlag`. The `deadline` corresponds to the virtual deadline $V(d_i)$ from the paper. The `slice` field represents the length of a request, r, used to calculate deadline and to determine when a request is fulfilled. The `vruntime` and `vlag`, explained previously, are the core of the Linux implementation of EEVDF; they are used to track task eligibility. In Linux, just as in the EEVDF theory, a task is eligible when it has non-negative lag.

In the Linux implementation of the EEVDF algorithm, clients may leave and join the competition at any time regardless of their lag values. When a client with positive lag leaves the competition, that lag is remembered indefinitely and the client will rejoin the competition with the same positive lag. This way, if the client is owed time, it will be honored. Clients that leave the competition with negative lag use a "decaying" mechanism; their lag returns to 0 (is "forgiven") at the rate that virtual-time ticks. This mechanism allows long sleeping tasks eventually have their negative lag reset without being vulnerable to exploits where clients leave with negative lag and immediately rejoin. This mechanism is implemented as follows: when a task with negative lag leaves the system, it remains on the run-queue but is marked for "deferred dequeue" and is not schedulable. This process cannot be chosen to execute, but it will still have its lag reduced proportional to the rate of virtual time. When its lag becomes non-negative, it will be removed from the run-queue by the scheduler [14].

One important observation is that in the Linux implementation of EEVDF, all operations are discreet, floating point rather than idealized operations with infinite precision. This means that the Linux implementation of EEVDF is necessarily an approximation of the ideal scheduler. For example, the floating point calculations used to determine a client's share based on its weight, may cause the weights in the system to not perfectly sum to 1. In fact, the weights in

the system are scaled down in many computations so that values do not overflow across long executions.

The factors that influence a client's virtual deadline, and thus priority under EEVDF, are its weight, w_i, and its request size, r. Recall that (13) gives

$$V(d) = V(e) + \frac{r}{w_i}\,.$$

So, clients which make larger requests and have smaller weights will have larger virtual deadlines and be scheduled generally behind clients which make smaller requests and have larger weights. These are the tunables that a client may adjust to influence its responsiveness.

In the Linux EEVDF implementation, the concept of weight has been mapped to the Linux concept of "niceness." Nicer tasks, those with a larger niceness value, are mapped to a lower weight. Conversely, tasks with smaller niceness values are mapped to a higher weight. Then, nicer processes will have longer deadlines and be serviced behind processes with lower niceness values [12]. Clients with lower weights (nicer tasks) have a smaller share of the system and will receive less time on the CPU than clients with higher weights (less nice tasks).

The second tunable a task has to influence its treatment by the EEVDF scheduler is its request size. Tasks which issue longer requests will be scheduled behind tasks which issue shorter requests in general, allowing tasks which issue shorter requests to enjoy lower latency. Importantly, and unlike adjusting w_i as a responsiveness parameter, tasks which issue multiple shorter requests will still receive the same proportional share of the processor as tasks which issue fewer longer requests. In this way, EEVDF offers latency control without compromising fairness.

In practice, Linux tasks have limited mechanisms to influence their r value. They are able to request a specific time slice using the `sched_setattr()` system call [15]. If this is not done, the request length will be the system parameter sched_base_slice_ns, which maps directly to the quantum q from the literature [13].

There have been discussions around other mechanisms to allow Linux tasks to request a specific time slice length. A new task parameter called a *latency-nice* value has been proposed as a patch to the EEVDF scheduler which would determine the request length of a task in a similar manor that the niceness value of a task currently determines its weight. In this patch, a lower latency-nice value value would mean tighter latency requirements and shorter time slice [8], [16]. This patch was proposed in 2022 and has yet to be adopted.

6 EEVDF as a Real-Time Algorithm

In a real-time context, the classic task model is *tasks* which issue schedulable entities called *jobs* to be scheduled on some resource—here we will say the CPU. Each job has an absolute deadline d associated with it. We map the notion of a

real-time *task* to an EEVDF *client* and the notion of a job to an EEVDF client making a *request*.

A *hard real-time* system one in which each job is required to be fulfilled by its deadline. These are often safety critical systems where timing must be predictable *e.g.*, airplane software, self-driving cars, pacemakers. A *soft real-time* system one in which each job must be fulfilled in a bounded amount of time after its deadline. These are systems where timing is important, but failure to miss a deadline is not catastrophic *e.g.*, video/audio streaming, conference calling, network routing devices.

The first remark we make about EEVDF for real-time is that it is generally suitable only for soft real-time contexts. The algorithm provides timing guarantees regarding deadlines that are relative to other weights in the system. For example, a job that releases in a system without many other clients will have an earlier deadline than that same job released in a system whose existing clients have a larger sum of weights. This is behavior is unpredictable, making the algorithm unsuited for hard real-time contexts without further bookkeeping like *admission control*, when a real-time system confirms a task does not compromise the real-time performance of existing tasks in the system every time a new task tries to enter. This incompatibility is logical, given the algorithm was designed with fairness in mind and not necessarily for real-time contexts. does not provide the absolute timing guarantees needed for hard real-time contexts.

The second remark we make in this section is that EEVDF is well-suited for soft real-time tasks specifically in Linux. Because of the sched_class heirarchy discussed in Figure 6, running real-time processes under real-time scheduling classes in Linux can negatively impact the performance of the processes running in non-real-time scheduling classes. Therefore, it becomes rather convenient to be able to make even relative timing guarantees for tasks in the non-real-time scheduling classes, a property which the best-effort CFS lacked. In fact, both real-time and non-real time processes look the same to the EEVDF scheduler; all tasks are guaranteed to be allocated proportionally fair time on the CPU.

7 Fairness Analysis

In this section, we present useful definitions as well major results pertaining to the EEVDF algorithm. The proofs for the theorems in this section can be found in the Appendix.

Definition 2. *A **steady** system (**S-system** for short) is a system in which the lag of any client that joins, or leaves the competition is zero.*

Definition 3. *A **pseudo-steady** system (**PS-system** for short) is a system in which the lag of any client that joins is zero, and the lag of any client that leaves is non-negative. Moreover, when a client with positive lag leaves, the value of virtual-time is updated according to **R1**.*

Definition 4. *A **dynamic** system (**D-system** for short) is a system in which there are no restrictions on the lag of clients that leave or join. Moreover, when*

a client leaves the system, the value of virtual-time is updated according to ***R1*** *and when a client joins the system, the value of virtual-time is updated according to* ***R2****.*

Definition 5. *A* ***pending*** *request at time t is a request which has been issued by a client, but not yet fulfilled.*

Theorem 1. Let d be the deadline of the current request issued by τ_k in an S-system with quantum q and let f be the actual time when this request is fulfilled. Then,

1. the request is fulfilled no later than $d + q$, *i.e.,* $f < d + q$ and
2. if $f > d$, for any time $t \in [d, f)$, $\text{lag}_k(t) < q$.

Theorem 2. Let r be the size of the current request issued by client τ_k in an S-system with quantum q. Then, the lag of τ_k at any time t while the request is pending is bounded as follows:

$$-r < \text{lag}_k(t) < \max(r, q)$$

Moreover, these bounds are asymptotically tight.

Corollary 1. If no request of τ_k is larger than a time quantum, then at any time t its lag is bounded as follows:

$$-q < \text{lag}_k(t) < q\,.$$

Proof. As stated in [2], this corollary follows directly from Theorems 1 and 2. □

Definition 6. *For any S-system of $n > 0$ clients in which each τ_i is assigned a positive weight w_i, a* ***proportional-share algorithm*** *aims to allocate each task proportionally fair time on the resource, i.e.,* $\int_{t_1}^{t_2} \frac{w_i}{\sum_{\tau_j \in A(t)} w_j} \mathrm{d}t$ for all $[t_1, t_2)$ where τ_i active.

Theorem 3. Let r be the size of the current request issued by client τ_k in a PS-system with quantum q. Then, the lag of τ_k at any time t while the request is pending is bounded as follows,

$$-r < \text{lag}_k(t) < \max(R_{max}, q)\,,$$

where R_{max} is the maximum duration of any request issued by any client in the system.

Theorem 4. For any S-system with time quantum q and any proportional-share algorithm, the lag of any client is bounded by $-q$ and q.

Corollary 2. EEVDF is optimal in the class of proportional-share algorithms.

Proof. This important corollary follows from Corollary 1 and Theorem 4. □

8 Conclusions

In this work, we gave a complete description of the EEVDF proportional-share scheduling algorithm, including an overview of various existing implementations for the handling of dynamic systems. We provided a discussion on the current state of EEVDF in the Linux kernel. We provided revised and expanded proofs for results describing the lag bounds of a client in an S-system, specifically that the lag of any client at any time is bounded by the length of a quantum q if we assume that no requests issued by clients exceed the length of q. Further, we provided proofs that the EEVDF algorithm is work-conserving and optimal in the class of fair schedulers.

Further work on the EEVDF algorithm includes the construction of a precise model of Linux's multiprocessor implementation of EEVDF in order to prove lag bound results.

References

[1] I. Stoica and H. Abdel-Wahab, "Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation," Department of Computer Science, Old Dominion University, Technical Report TR-95-22, Nov. 1995.

[2] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Dec. 1996.

[3] S. Baruah, J. Gehrke, and C. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Proceedings of 9th International Parallel Processing Symposium*, 1995, pp. 280–288.

[4] A. Srinivasan and J. H. Anderson, "Optimal rate-based scheduling on multiprocessors," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 189–198. [Online]. Available: https://doi.org/10.1145/509907.509938

[5] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: flexible proportional-share resource management," in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '94. USA: USENIX Association, 1994, p. 1–es.

[6] C. Wong, I. Tan, R. Kumari, J. Lam, and W. Fun, "Fairness and interactive performance of o(1) and cfs linux kernel schedulers," in *2008 International Symposium on Information Technology*, vol. 4, 2008, pp. 1–8.

[7] Community, Zildj1an, and J. Corbet, "EEVDF scheduler documentation," https://github.com/torvalds/linux/tree/master/Documentation/scheduler/sched-eevdf.rst, 2024, accessed: 2025-11-30.

[8] J. Corbet, "An EEVDF CPU scheduler for Linux," *LWN.net*, Mar. 2023, accessed: 2026-03-15. [Online]. Available: https://lwn.net/Articles/925371/

[9] L. community, "Linux kernel source code: `kernel/sched/`," https://github.com/torvalds/linux/tree/master/kernel/sched, accessed: 2025-11-30.

[10] corbet, "Kernel release status," https://lwn.net/Articles/949664/, Nov. 2023, accessed: 2025-11-30.

[11] Community, "Linux documentation: Documentation/scheduler/sched-design-cfs.rst," https://github.com/torvalds/linux/tree/master/Documentation/scheduler/sched-design-CFS.rst, accessed: 2025-11-30.

[12] F. Zwettler, "Implementation of an optimized Linux task scheduler for gateway applications utilizing the sched_ext subsystem in rust and ebpf," Nuremberg, Germany, October 2025.

[13] L. Torvalds and the Linux community, "Linux kernel," accessed: 2025-11-30. [Online]. Available: https://github.com/torvalds/linux

[14] J. Corbet, "Completing the EEVDF scheduler," https://lwn.net/Articles/969062/, April 2024, accessed: 2025-11-30.

[15] The Linux Kernel Developers, "EEVDF scheduler—the Linux kernel documentation," https://docs.kernel.org/scheduler/sched-eevdf.html, 2024, accessed: 2025-02-14.

[16] J. Corbet, "Improved response times with latency nice," *LWN.net*, Mar. 2022, accessed: 2026-03-15. [Online]. Available: https://lwn.net/Articles/887842/

Lemma 1. For all clients τ_i and times $t < t'$, if τ_i executes throughout $[t, t')$, then $\text{lag}_i(t) \geq \text{lag}_i(t')$ and if τ_i does not execute in $[t, t')$, then $\text{lag}_i(t) < \text{lag}_i(t')$.

Proof. Take client τ_i and times $t < t'$. Assume without loss of generality that all the weights in the system remain constant during $[t, t')$ since any $[t, t')$ with non-constant weights can be broken into a finite number of intervals with constant weights.

By (6) $\text{lag}_i(t') = \text{lag}_i(t) + S_i(t, t') - s_i(t, t')$. By our assumption of constant weights, we then have $\text{lag}_i(t') = \text{lag}_i(t) + (t - t')f_i(t) - s_i(t, t')$.

Case 1. τ_i executes through $[t, t')$. Then, $s_i(t, t') = t' - t$, and by (3),

$$\text{lag}_i(t') = \text{lag}_i(t) + (t' - t)f_i(t) - (t' - t) \leq \text{lag}_i(t)\,.$$

Case 2. τ_i does not execute in $[t, t')$. Then, $s_i(t, t') = 0$, and by (3),

$$\text{lag}_i(t') = \text{lag}_i(t) + (t' - t)f_i(t) > \text{lag}_i(t)\,.$$

□

Lemma 2. *If a client τ_i is active at time t and $\text{lag}_i(t) \geq 0$, then τ_i is eligible.*

Proof. Because τ_i is active at t, it has a pending request r_i^k at t such that $t_i^k \leq t$. Let $e_i^k = e$. Our proof obligation is to show $e \leq t$.

From the lemma statement, $\text{lag}_i(t) \geq 0$, which by (6) implies $S_i(t_i^0, t) - s_i(t_i^0, t) \geq 0$, and hence $S_i(t_i^0, t) - s_i(t_i^0, t_i^k) - s_i(t_i^k, t) \geq 0$. By (10), we have $S_i(t_i^0, t) - S_i(t_i^0, e) - s_i(t_i^k, t) \geq 0$.

If we assume, for the sake of contradiction, that $e > t$, from (4) we find $S_i(t_i^0, t) < S_i(t_i^0, e)$, which implies $S_i(t_i^0, t) - S_i(t_i^0, e) - s_i(t_i^k, t) < 0$, a contradiction. Hence, $e \leq t$, as intended. □

Lemma 3. *At any time t, if $|A(t)| \geq 1$, then system contains at least one client with non-negative lag.*

Proof. Take time t such that $|A(t)| \geq 1$. By (17), $\sum_{i \in A(t)} \text{lag}_i(t) = 0$. A contradiction arises if we assume the converse of the lemma statement, that $\text{lag}_i(t) > 0$ for all $i \in A(t)$. □

Lemma 4. *The EEVDF algorithm is work-conserving, i.e., the resource is not idle if there is an active client.*

Proof. Assume there is at least one active client in the system at a time t. We are obliged to show that the resource is not idle at t. First, by Lemma 3, there exists at least one active client with non-negative lag at t. By Lemma 2, this client has an eligible request at t. Then, the set of eligible requests at time t is non-empty and the member with the the earliest virtual deadline will be scheduled by EEVDF (Definition 1) at time t and the resource will not be idle. □

Lemma 5. *If τ_i has an active request r_i^k at time t with $d_i^k \leq t$, then $\text{lag}_i(t) > 0$.*

Proof. Take τ_i with active request r_i^k at time t, with $d_i^k \leq t$.

Then, r_i^k is not fulfilled at time t, which implies that τ_i has received less than its requested time on the processor since its release, *i.e.,*

$$s_i(t_i^k, t) < r\,. \tag{20}$$

Starting with (12),

$$\begin{aligned}
S_i(e_i^k, d_i^k) &= r \\
&\Rightarrow \{d_i^k \leq t,\ (4)\} \\
S_i(e_i^k, t) &\geq r \\
&= \{(20)\} \\
S_i(e_i^k, t) &\geq r > s_i(t_i^k, t) \\
&\Rightarrow \{\text{algebra}\} \\
S_i(e_i^k, t) &> s_i(t_i^k, t) \\
&= \{(10)\} \\
S_i(t_i^0, e_i^k) + S_i(e_i^k, t) &> s_i(t_i^0, t_i^k) + s_i(t_i^k, t) \\
&= \{\text{combine}\} \\
S_i(t_i^0, t) &> s_i(t_i^0, t) \\
&= \{(6)\} \\
\text{lag}_i(t) &> 0\,.
\end{aligned}$$

□

Theorem 1. Let d be the deadline of the current request issued by τ_k in an S-system with quantum q and let f be the actual time when this request is fulfilled. Then,

1. the request is fulfilled no later than $d + q$, *i.e.,* $f < d + q$ and
2. if $f > d$, for any time $t \in [d, f)$, $\text{lag}_k(t) < q$.

Proof. As in the paper body, $A(t)$ is the set of active clients at time t. We define the following partition of $A(t)$.

Definition 7. *For any time t such that τ_k has an active request, $B(t)$ contains all clients that have an active request at time t with a deadline earlier than or at d and $C(t)$ contains all clients that have an active request at time t with a deadline strictly after d.*

Note, $\tau_k \in B(t)$ for all t, τ_k active. Also note that since a client can have at most one request at a time, $B(t)$ and $C(t)$ are disjoint; $B(t) = A(t) \setminus C(t)$.

Let $t_1 < d$ be the latest time the request of client τ_ℓ with a deadline greater than d starts to receive a quantum. Note, $\tau_\ell \in C(t)$. We asses two cases: that t_1 exists and that it doesn't.

Case 1. t_1 exists.

First, we assume $f > d$, otherwise the theorem is trivially true.

Claim 1:

$$\forall \tau_i \in B(t_1),\, \mathrm{lag}_i(t_1) < 0.$$

Proof Claim 1. Take $\tau_i \in B(t_1)$. Assume for the sake of contradiction that $\exists \tau_i \in B(t_1)$ such that $\mathrm{lag}_i(t_1) \geq 0$. By Lemma 2, this means that the pending request of τ_i is eligible at t_1. Because all requests in $B(t)$ have a deadline before requests in $C(t)$ for all t, the quantum at t_1 would be given to τ_i's request instead of τ_ℓ's, causing a contradiction with the case assumption that t_1 exists. So, we must have $\mathrm{lag}_i(t_1) < 0$ and we have proved Claim 1.

From Claim 1, we get $\sum_{\tau_i \in B(t_1)} \mathrm{lag}_i(t_1) < 0$. By $C(t_1) = A(t_1) \setminus B(t_1)$ and (17), we then also must have

$$\sum_{\tau_i \in C(t_1)} \mathrm{lag}_i(t_1) > 0. \tag{21}$$

Proving Part 2) for Case 1

We have already assumed the premise of 2), *i.e.,* $f > d$. Take time $T \in [d, f)$. Our proof obligation is to bound the lag at time T by the length of a quantum, q.

Claim 2:

$$\sum_{\tau_i \in C(T)} \mathrm{lag}_i(T) > -q\,.$$

Proof Claim 2.

We begin with the following Facts.

1. Excepting τ_ℓ, no other client with a deadline greater than d receives any service time in $[t_1, T]$. This is from the definition of t_1.

2. Since the maximum size of a time quantum is q, the service time received by client ℓ in $[t_1, T]$ is at most q.

3. Since we are considering a steady system, Definition 2, all clients that leave or join the competition have zero lags.

Let L be the set of clients in $C(t_1)$ that leave the system at any point in $(t_1, T]$. Let J be the set of clients in $C(t_1)$ that join the system at any point in $(t_1, T]$. Then,

$$C(T) = (C(t_1) \setminus L) \cup J\,. \tag{22}$$

By Fact 3, all the clients in L left with zero lag and

$$\sum_{\tau_i \in L} \mathrm{lag}_i(T) = 0\,. \tag{23}$$

By Fact 3, all the clients in J joined with zero lag and by Fact 1, do not receive any service time in $[t_1, T]$. By (6),

$$\sum_{\tau_i \in J} \text{lag}_i(T) > 0\,. \tag{24}$$

From (21),

$$\begin{aligned}
&\sum_{\tau_i \in C(t_1)} \text{lag}_i(t_1) > 0 \\
&\qquad \Rightarrow \{(4)\} \\
&\sum_{\tau_i \in C(t_1)} \text{lag}_i(t_1) + S_i(t_1, T) > 0 \\
&\qquad = \{\text{Facts 1, 2} \Rightarrow \sum_{\tau_i \in C(t_1)} s_i(t_1, T) \leq q\} \\
&\sum_{\tau_i \in C(t_1)} \text{lag}_i(t_1) + S_i(t_1, T) - s_i(t_1, T) > -q \\
&\qquad = \{(6)\} \\
&\sum_{\tau_i \in C(t_1)} S_i(t_i^0, t_1) - s_i(t_i^0, t_1) + S_i(t_1, T) - s_i(t_1, T) > -q \\
&\qquad = \{(6)\} \\
&\sum_{\tau_i \in C(t_1)} \text{lag}_i(T) > -q\,.
\end{aligned} \tag{25}$$

From (23), (24), and (25), we have

$$\begin{aligned}
&\sum_{\tau_i \in L} \text{lag}_i(T) + \sum_{\tau_i \in J} \text{lag}_i(T) + \sum_{\tau_i \in C(t_1)} \text{lag}_i(T) > -q \\
&\qquad = \{(22)\} \\
&\sum_{\tau_i \in C(T)} \text{lag}_i(T) > -q\,.
\end{aligned}$$

This concludes the proof of Claim 2.

As a result of Claim 2, by (17), we have

$$\sum_{\tau_i \in B(T)} \text{lag}_\text{i}(\text{T}) < \text{q}\,. \tag{26}$$

Claim 3: All clients in $B(T)$ have positive lag.

Proof Claim 3. Take $\tau_i \in B(T)$. τ_i has an active request at T with a deadline $d_i^k \leq d < T$ by Definition 7. Then, by Lemma 5, $\text{lag}_i(T) \geq 0$, proving Claim 3.

By (26) and Claim 3, we find $\text{lag}_i < q$ for all $\tau_i \in B(T)$. Because $\tau_k \in B(T)$, we conclude:

$$\text{lag}_k(T) < q\,.$$

This proves part 2) of Theorem 1. At every point $T \in [d, f)$ (these are points where τ_k is unfulfilled) we may bound $\text{lag}_k(T)$ by the length of a quantum.

Proving Part 1) for Case 1 As a reminder, we have $f > d$. This implies that τ_k is active at time d and, by Definition 7, that $\tau_k \in B(d)$. Then, to prove part 1) for the case where t_1 exists, our proof obligation will be to show that at time d, all requests in $B(d)$ will be fulfilled in less than q time units.

Claim 4: All clients in $B(d)$ are eligible for all times $t \geq d$.

Proof Claim 4. Take $\tau_i \in B(d)$ and $t \geq d$. Then, τ_i has an active request at d with a deadline before or at t. By Lemma 5, $\text{lag}_i(t) > 0$. A client is eligible when it has non-zero lag, so we have proved Claim 4.

From Claim 4 and because the deadlines of all clients in $B(d)$ are less than the deadlines of all clients in $C(d)$, by the EEVDF policy, all requests in $B(d)$ are serviced before any request in $C(d)$. We know also that no work can join $B(d)$ after point d and that all clients in $B(d)$ are eligible at all times at and after d, so we know that clients in $B(d)$ will be serviced continuously from time d.

From (26) and $T \in [d, f)$, we have $\sum_{\tau_i \in B(d)} \text{lag}_i(d) < q$ and from Lemma 5 we have that $\text{lag}_i(d) > 0$ for each $\tau_i \in B(d)$.

Then, it will take less than q time units from d to fulfill the requests by clients in $B(d)$, fulfilling our proof obligation for Part 1), Case 1.

Case 2. t_1 does not exist.

In other words, this case states that there is no client with a deadline greater than d that starts receiving a time quantum before d.

In this case, we will prove that τ_k's request must in fact be fulfilled by its deadline d.

Claim 5:

$$\sum_{\tau_i \in B(d)} \text{lag}_i(d) \leq 0\,.$$

Proof Claim 5. In the case that $C(d)$ is empty, we have $(\sum_{\tau_i \in C(d)} \text{lag}_i(d) = 0) \Rightarrow (\sum_{\tau_i \in B(d)} \text{lag}_i(d) = 0)$ by (17), proving Claim 5 in this case.

Now we assume the $C(d)$ is non-empty. Take $\tau_i \in C(d)$. By Definition 7, τ_i has an active request r_i^n at time d with a deadline $d_i^n \geq d$. By case statement,

τ_i does not execute in $[t_i^k, d)$. Beginning with the (6),

$$\begin{aligned}
\mathrm{lag}_i(d) &= S_i(t_i^0, d) - s_i(t_i^0, d) \\
&= \{(6)\} \\
\mathrm{lag}_i(d) &= \mathrm{lag}_i(t_i^n) + S_i(t_i^n, d) - s_i(t_i^n, d) \\
&= \{\text{S-system}\} \\
\mathrm{lag}_i(d) &= S_i(t_i^n, d) - s_i(t_i^n, d) \\
&= \{s_i(t_i^n, d) = 0\} \\
\mathrm{lag}_i(d) &= S_i(t_i^n, d) \\
&= \{S_i(t_i^n, d) \geq 0\} \\
\mathrm{lag}_i(d) &\geq 0\,.
\end{aligned}$$

From this, we have

$$\sum_{\tau_i \in C(d)} \mathrm{lag}_i(d) \geq 0\,.$$

From (17), we have

$$\sum_{\tau_i \in B(d)} \mathrm{lag}_i(d) \leq 0\,,$$

which proves Claim 5.

We now prove that τ_k's request must be fulfilled before its deadline d by contradiction. Consider Claim 7:

Claim 7: If $B(d)$ non-empty,

$$\sum_{\tau_i \in B(d)} \mathrm{lag}_i(d) > 0\,.$$

Proof Claim 7. Take $\tau_i \in B(d)$. By Definition 7, τ_i has an active request r_i^n at time d with a deadline $d_i^n \leq d$. Then, by Lemma 5, $\mathrm{lag}_i(d) > 0$. Then, $\sum_{\tau_i \in B(d)} \mathrm{lag}_i(d) > 0$, proving Claim 7.

So, we have Claim 7 contradicting Claim 6. Thus, $B(d)$ must be empty. Hence, there is no client with a deadline before or at d with an unfulfilled request at time d. Then, τ_k is fulfilled before time d, proving the lemma for Case 2.

□

Theorem 2. Let r be the size of the current request issued by client τ_k in an S-system with quantum q. Then, the lag of τ_k at any time t while the request is pending is bounded as follows:

$$-r < \mathrm{lag}_k(t) < \max(r, q)$$

Moreover, these bounds are asymptotically tight.

Proof. Take client τ_k with a pending request at time t, r_k^n of size r that has a deadline at time d.

Request r_k^n pending at time t can be expressed mathematically as

$$s_k(t_k^n, t) < r\,. \tag{27}$$

We first prove the LHS of the equation: $-r < \text{lag}_k(t)$.

Claim 1: $S_k(t_k^0, t) - S_k(t_k^0, e_k^n) \geq 0$.

Proof Claim 1. Because r_k^n pending at t, $e_k^n \leq t$. Then, by (4), $S_k(t_k^0, t) \geq S_k(t_k^0, e_k^n)$, which is equal to Claim 1.

To prove the LHS, we start with the lag definition (6)

$$\begin{aligned}
S_k(t_k^0, t) - s_k(t_k^0, t) &= \text{lag}_k(t) \\
&= \{\text{algebra}\} \\
S_k(t_k^0, t) - s_k(t_k^0, t_k^n) - s_k(t_k^n, t) &= \text{lag}_k(t) \\
&= \{(10)\} \\
S_k(t_k^0, t) - S_k(t_k^0, e_k^n) - s_k(t_k^n, t) &= \text{lag}_k(t) \\
&= \{\text{Claim 1}\} \\
0 - s_k(t_k^n, t) &\leq \text{lag}_k(t) \\
&= \{(27)\} \\
-r &< \text{lag}_k(t)\,,
\end{aligned}$$

completing the proof of the LHS.

Now, we move to prove the RHS of the equation: $\text{lag}_k(t) < \max(r, q)$.

Since as soon as τ_k has positive lag, τ_k is eligible (from Lemma 2), we only consider $t \geq e_k^n$. We consider two cases.

Case 1. $t < d$.

From the case statement and (4), we have

$$S_k(t_k^0, t) < S_k(t_k^0, d)\,. \tag{28}$$

We also have

$$s_k(t_k^0, e_k^n) \leq s_k(t_k^0, t)\,, \tag{29}$$

since $e_k^n \leq t$ and $s_i(t, t')$ is a non-negative quantity for all $t \leq t'$.

Then, using the lag definition (6),

$$\begin{aligned}
\mathrm{lag}_k(t) &= S_k(t_k^0, t) - s_k(t_k^0, t) \\
&\Rightarrow \{(28)\} \\
\mathrm{lag}_k(t) &< S_k(t_k^0, d) - s_k(t_k^0, t) \\
&\Rightarrow \{(29)\} \\
\mathrm{lag}_k(t) &< S_k(t_k^0, d) - s_k(t_k^0, e_k^n) \\
&= \{\text{algebra}\} \\
\mathrm{lag}_k(t) &< S_k(t_k^0, e_k^n) + S_k(e_k^n, d) - s_k(t_k^0, e_k^n) \\
&= \{(10)\} \\
\mathrm{lag}_k(t) &< s_k(t_k^0, t_k^n) + S_k(e_k^n, d) - s_k(t_k^0, e_k^n) \\
&= \{(12)\} \\
\mathrm{lag}_k(t) &< s_k(t_k^0, t_k^n) - s_k(t_k^0, e_k^n) + r \\
&\Rightarrow \{(t_k^n \leq e_k^n) \Rightarrow s_k(t_k^0, t_k^n) \leq s_k(t_k^0, e_k^n)\} \\
\mathrm{lag}_k(t) &< r \,,
\end{aligned}$$

providing an upper bound for $\mathrm{lag}_k(t)$ in the case that $t < d$.

Case 2. $t \geq d$.

Call the time that r_k^n is fulfilled f. From τ_i pending at $t \geq d$, we have $f > d$. By Theorem 1, we have $\mathrm{lag}_k(t) < q$ for all $t \in [d, f)$. We do not need to consider $t \geq f$ because τ_k has a pending request at t.

We combine the results from both cases to complete the proof of the right hand side. We have $\mathrm{lag}_k(t) < r$ for Case 1, $t < d$ and $\mathrm{lag}_k(t) < q$ for Case 2, $t \geq d$. So, for any time t where the request is pending, we have $\mathrm{lag}_k(t) < \max(r, q)$, as intended.

Asymptotically tight bounds. We now demonstrate the tightness of the bounds expressed in Theorem 2 by constructing examples in which the bounds are approached.

First, the lower bound, *i.e.,* $-r < \mathrm{lag}_k(t)$. We consider a system consisting of only two clients, τ_k and τ_j, with $w_k << w_j$, which both release requests at time t_0. We choose r_k, r_j such that $V(d_k) < V(d_j)$ - the virtual deadline of τ_k is before the virtual deadline of τ_j. This means that under EEVDF, τ_k's request will execute first. Using definitions (6) and (5), we have

$$\mathrm{lag}_k(r_k) = S_k(t_0, t_0 + r_k) - s(t_0, t_0 + r_k) = \frac{w_k}{w_k + w_j}(r_k) - r_k \,.$$

We let $w_j \to \infty$, noting that for all values of w_j we are able to find appropriate r_k, r_j to satisfy $V(d_k) < V(d_j)$. Then,

$$\mathrm{lag}_k(r_k) = \frac{w_k}{w_k + w_j}(r_k) - r_k \to -r_k \,.$$

So, we have found a system and a time t such that $\text{lag}_k(t) \to -r$, proving that the lower bound in Theorem 2 cannot be any higher; it is tight.

Now, we prove the tightness of the upper bound, *i.e.,* $\text{lag}_k(t) < \max(r, q)$.

First, consider the case $r < q$. Again, we will take a system with only two clients, τ_k and τ_j and quantum q relatively large. Assume τ_j is the only client active at time t_0, and thus it is scheduled at time t_0. Assume just after time t_0, at $t_0 + \epsilon$, τ_k issues a request of length $r_k < q$. The intuition behind this example is that since τ_j has already been scheduled, it must execute for a full quantum q before τ_k with an earlier deadline can be scheduled.

To this end, take $w_k >> w_j$. Let $w_k \to \infty$. Then by definitions (6) and (5),

$$\text{lag}_k(t_0 + \epsilon + q) = S_k(t_0 + \epsilon, t_0 + \epsilon + q) - s_k(t_0 + \epsilon, t_0 + \epsilon + q) = \frac{w_k}{w_k + w_j}(q) \to q\,.$$

Next, consider case $q \leq r$.

Consider a system with n clients. Clients $\tau_1, \ldots, \tau_{n-1}$ have $w_i = \frac{1}{2}$ and each release requests of length 1 at time 0. Client τ_n has a $w_n = 1$ and releases a request of length 2 at time 0. Take $q = 1$.

When two clients τ_i, τ_j have the same virtual deadline, the scheduler will break ties by scheduling τ_i if $i < j$.

We will see that the lag of client τ_n approaches its request size, $2 > q$, in the limit of the number of clients, n.

Using (13), we have $V(d_i^1) = V(e_i^1) + \frac{r_i^1}{w_i} = 0 + \frac{1}{0.5} = 2$ for $i \neq n$. Again by (13), we have $V(d_n^1) = V(e_n^1) + \frac{r_n^1}{w_n} = 0 + \frac{2}{1} = 2$.

By the tie-break rule, we have that $\tau_1, \ldots, \tau_{n-1}$ are scheduled before τ_n, each for one time quantum. So, by (6), we find the following expression for the lag of τ_n just before it is scheduled

$$\begin{aligned}
\text{lag}_n(q(n-1)) =& S_n(0, q(n-1)) - s_n(0, q(n-1)) \\
=& \{q = 1\} \\
\text{lag}_n(n-1) =& S_n(0, (n-1)) - s_n(0, (n-1)) \\
=& \{\tau_1, \ldots, \tau_{n-1} \text{ scheduled before } \tau_n\} \\
\text{lag}_n(n-1) =& S_n(0, (n-1)) \\
=& \{(5)\} \\
\text{lag}_n(n-1) =& (n-1)(\frac{1}{1 + \frac{1}{2}n}) \\
=& \{\text{algebra}\} \\
\text{lag}_n(n-1) =& \frac{2n-2}{n+2} \xrightarrow{n \to \infty} 2\,.
\end{aligned}$$

So, we have found systems and times t such that $\text{lag}_k(t) \to r$ and $\text{lag}_k(t) \to q$, proving that the upper bound in Theorem 2 cannot be any lower; it is tight. □

Theorem 3. Let r be the size of the current request issued by client τ_k in a PS-system with quantum q. Then, the lag of τ_k at any time t while the request

is pending is bounded as follows,

$$-r < \text{lag}_k(t) < \max(R_{max}, q)\,,$$

where R_{max} is the maximum duration of any request issued by any client in the system.

Proof. The proof for this theorem is found in [2]. □

Theorem 4. For any S-system with time quantum q and any proportional share algorithm, the lag of any client is bounded by $-q$ and q.

Proof. The proof of this theorem is very similar to that in [1] and essentially follows from the fact that time on the resource must be allocated in discreet quanta q.

First, we assert the existence of a proportional share algorithm where the lag of any client is bounded by $-q$ and q in a steady system by Corollary 1; we take EEVDF and modify any request that is larger than a time quantum to be multiple requests, each less than or equal to a time quantum.

Next, we prove that these bounds cannot be any tighter for any proportional share algorithm.

First, we prove the tightness of the upper bound. For the sake of contradiction, assume there exists an algorithm such that the lag of all clients at any time may be bounded above by $q - \epsilon$, where $\epsilon > 0$. Take a system of n clients with equal weights. In any system, there must exist a τ_k that is scheduled after all other clients. So, τ_k is scheduled at or after $t = q(n-1)$. Then, by (6),

$$\begin{aligned}
\text{lag}_k(q(n-1)) =& S_n(0, q(n-1)) - s_n(0, q(n-1)) \\
=& \{\tau_k \text{ not scheduled before } q(n-1)\} \\
\text{lag}_k(q(n-1)) =& S_n(0, q(n-1)) \\
=& \{(5)\} \\
\text{lag}_k(q(n-1)) =& \frac{1}{n}(q(n-1)) = q - \frac{q}{n}\,.
\end{aligned}$$

Take $n \in \mathbb{Z}$, $n > \frac{q}{\epsilon}$. Then, $\text{lag}_k(q(n-1)) > q - \epsilon$, which is a contradiction with our assumption.

Next, we prove the tightness of the lower bound. For the sake of contradiction, assume that there exists an algorithm such that the lag of all clients at any time may be bounded below by $-q + \epsilon$, where $\epsilon > 0$. Take a system of n clients with equal weights. In any system, there must exist a τ_k that is scheduled in the first time quantum. By (6),

$$\begin{aligned}
\text{lag}_k(q) =& S_n(0, q) - s_n(0, q) \\
=& \{\tau_k \text{ scheduled first quantum}\} \\
\text{lag}_k(q) =& S_n(0, q) - q \\
=& \{(5)\} \\
\text{lag}_k(q) =& \frac{q}{n} - q\,.
\end{aligned}$$

Take $n \in \mathbb{Z}$, $n > \frac{q}{\epsilon}$. Then, $\text{lag}_k(q(n-1)) < -q + \epsilon$, which is a contradiction with our assumption.

□