PBS: A Unified Priority-Based Scheduler *

Hanhua Feng
Dept. of Computer Science
Columbia University
hanhua@cs.columbia.edu

Vishal Misra
Dept. of Computer Science
Columbia University
misra@cs.columbia.edu

Dan Rubenstein
Dept. of Electrical Engineering
Columbia University
danr@ee.columbia.edu

ABSTRACT

Blind scheduling policies schedule tasks without knowledge of the tasks' remaining processing times. Existing blind policies, such as FCFS, PS, and LAS, have proven useful in network and operating system applications, but each policy has a separate, vastly differing description, leading to separate and distinct implementations. This paper presents the design and implementation of a *configurable* blind scheduler that contains a continuous, tunable parameter. By merely changing the value of this parameter, the scheduler's policy exactly emulates or closely approximates several existing standard policies. Other settings enable policies whose behavior is a hybrid of these standards. We demonstrate the practical benefits of such a configurable scheduler by implementing it into the Linux operating system. We show that we can emulate the behavior of Linux's existing, more complex scheduler with a single (hybrid) setting of the parameter. We also show, using synthetic workloads, that the best value for the tunable parameter is not unique, but depends on distribution of the size of tasks arriving to the system. Finally, we use our formulation of the configurable scheduler to contrast the behavior of various blind schedulers by exploring how various properties of the scheduler change as we vary our scheduler's tunable parameter.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes; F.2.2 [**Nonnumerical Algorithms and Problems**]: Sequencing and scheduling; G.3 [**Probability and Statistics**]: Queueing theory

General Terms

Measurement, Performance, Experimentation, Theory

*This work was supported in part by NSF grant CNS-0615126 and research gifts from Microsoft and IBM.

SIGMETRICS'07, June 12–16, 2007, San Diego, California, USA.

Keywords

PBS, FCFS, LAS, Queueing systems, Scheduling, Linux

1. INTRODUCTION

The performance of a system that must simultaneously process multiple tasks depends in large part on the *scheduling policy*, whose role is to decide how and when to process the various tasks waiting for service. Scheduling policies for operating and network systems have been studied for many years for a wide variety of *demands* that can occur in practice. The types of demands can vary considerably. For instance, multimedia applications require a predictable level of service to maintain a smoothness in their playout. In contrast, computer games require responsiveness, while machines used to perform general computations require small waiting times to completion.

One basic conclusion to be drawn from prior work is that one must take into account the type of demand the system is expected to handle when trying to find a good scheduler. Unfortunately, in practice, systems are often expected to handle a wide variety of demands, complicating the task of finding the "right" or "best" scheduling policy for that system. For instance, the scheduling policy in the Linux operating system is constantly changing throughout the 2.6 versions. We conjecture that these changes are attempts to tune the scheduler to do a "good" task across the constantly changing demands that an operating system is expected to face. There are many reasons why an operating system must work well for a large variety of demands. We present two possible reasons out of many:

- *Workload characteristics change.* A server may be used at one time as dedicated Linux firewall, processing many small tasks, and later be used as a computational server, processing a few long tasks. Even when a server's basic function is not altered, the workload it receives may change because it runs a new application. For instance, it is well-known that the installation of P2P file-exchange applications drastically changes a machine's workload[11].
- *Performance needs vary.* The same version of operating system is often used within a user's personal desktop, where responsiveness is required, in a server, where high processing throughput is paramount, and in a real-time processing system, where the percentage of real-time tasks that finish before their deadlines is the measure of interest [16].

This paper focuses specifically on the class of scheduling

policies that are *blind*. A blind scheduler is a scheduler that cannot use tasks' remaining processing times when scheduling tasks for processing. The scheduler can, however, utilize other attributes of a task, such as its arrival time and processing received thus far. First-Come First-Serve (FCFS), which prioritizes tasks according to their arrival time, Processor Sharing (PS), which splits processor share equally across all current tasks, and Least Attained Service (LAS) (also known as Foreground-Background Processor Sharing, FBPS and Feedback, FB) which prioritizes tasks that have received the least amount of service [24] are all examples of blind scheduling policies. Not only is the performance of these blind scheduling policies well-studied in theory [15], they are also widely implemented in today's networks and operating systems [27, 4]. In fact, almost all general-purpose operating systems such as Windows and Unix utilize blind scheduling policies. The reason for selecting blind scheduling policies stems from the availability of the task size information: operating systems often do not know a priori the entire processing demand of its active tasks, and networks frequently transport flows whose total size is unknown prior to completion.

Generally, each blind scheduling policy described in literature or used in practice has a description and implementation that is distinct from the other policies. Hence, to enable a system to gracefully adapt or tune to the current specific needs of the system, the system design would have to separately implement and maintain the various policies. Though this solution is viable, the additional maintenance expense makes it less practical, and (to our knowledge) is rarely done. More importantly, it is only possible to implement a discrete set of policies in this manner, and there are limited guidelines for how and when to use or switch between the various policies, or how to design a good hybrid policy (e.g., a policy in which tasks arriving earlier should receive greater share, but not exclusively, as is done by FCFS). Only very recently are people interested in developing configurable or parameterized scheduling policies [14, 18, 13].

This paper presents a *configurable policy*, named *Priority-based Blind Scheduling* (**PBS** for short), prioritizing tasks on not only their service received thus far but also their stay time in the system. PBS unifies *a few* common blind scheduling policies under a single umbrella and is easily tuned to accommodate for variations in demand. The policy contains a single tunable parameter, α, which, when set to specific values, enables the scheduler to closely approximate a number of well-known blind policies. Furthermore, the parameter can be set to any (in theory continuous) positive value, creating a smooth transition from one standard policy to another. This smooth transitioning also enables a continuum of hybrid policies that mix the various benefits of the well-known policies between which they lie. The unifying property of PBS provides theorists with a new tool to compare and contrast various (existing or novel) blind scheduling policies and, at the same time, provides practitioners with a more flexible scheduler that can be tuned on-line to optimize for current demands. Briefly, the contributions of this paper are:

- a description of PBS and its tunable parameter, α,
- simulation results demonstrating the ability of the scheduler to emulate or approximate both standard policies as well as their hybrids,
- analysis of the behavior of this blind scheduler as a function of α, giving a unified treatment of existing and novel (hybrid) blind scheduling polices, contrasting these policies' trade-offs in terms of performance and fairness,
- an implementation of PBS within a current Linux operating system that emulates its (significantly more complex) scheduler, and
- experiments using synthetic workloads to demonstrate that to minimize expected response time, the optimal value of α depends *strongly* on the distribution of task sizes, thus reaffirming the assertion that there is no single α to rule them all. Different scenarios require a different (hybrid) scheduler.

For our analytical contributions, we show that PBS approximates FCFS, PS, and LAS when its parameter, α, is respectively tuned (in the limit when applicable) to 0, 1, and ∞. We identify various theoretical properties of PBS that hold under different ranges of α. For instance, we prove that, as α increases, mean response time decreases (increases) for decreasing(increasing)-hazard-rate task-size distributions. In other words, for heavy-tailed tasks, by increasing α a system will further reduce the expected time to complete tasks, whereas decreasing α is desirable for light-tailed tasks. We show that, as this α increases, the variation of received service reduces, as does the total service received by tasks that arrive earlier. We also show that the novel hybrid policies that we propose overcome pathological problems of FCFS and LAS such as starvation, while at the same time approximating the desired behavior under normal conditions to an arbitrary degree of precision.

For our practical contributions, we show, using two different synthetic workloads, that different values of α in these two settings to minimize expected response time of arriving tasks. This result demonstrates the potential benefit of a tunable scheduler: the scheduler is not tied to a fixed policy, and if task sizes are not known, but the general distribution is known or can be estimated, it should be possible to add a control loop to tune α to minimize response time for those arrivals. The monotonicity property of response time with respect to α, given general properties of a task size distribution, indicates the problem of designing a well behaved controller should be feasible. We also describe the challenges and additional issues that arose as we transitioned PBS from a policy "in theory" to a practical Linux environment.

1.1 Related work

Although the PBS falls in the general framework proposed by Ruschitzka and Fabry [23], the prior analytic work on blind schedulers handles different policies in isolation. In [8], Coffman *et al.* study the PS policy and compare it to round robin. PS is extended in to Generalized Processor Sharing (GPS) [19] and Discriminated Processor Sharing (DPS) [9] to support weighted sharing. See Caprita *et al.* [7] for a more complete list of GPS variants and implementations. The LAS/FBPS/FB policy is first studied by Schrage [24], but has received significant recent attention for the case where the task sizes have a large coefficient of variation [20]. Several other blind scheduling policies, such as multi-level feedback-queue scheduling (MLFS) [27], Multi-level Processor-Sharing (MLPS) [15] and its special cases [1], and Preemptive Last-Come First-Serve (PLCFS) [2] have also been studied. For the sake of completeness,

we note that there is a significant body of work that studies non-blind policies. For instance, Schrage and Miller analyze the Shortest Remaining Processing Time (SRPT) policy and Shortest Jobs First (SJF) policy [26, 25]. Few parameterized scheduling policies are studied in the literature, including the early work by Mitrani and Hine [17]. Recently, Kherani and Núñez-Queija study TCP in relation to a parameterized scheduling policy based on attained service time, and the continuous version of this policy is subsequently analyzed by Padhy and Kherani [18], and Kherani [13]. Wierman et al.[29] analyze a class of non-blind policies, although this class is not parameterized.

Though much of work focuses on minimizing expected task response time, we are also concerned with other measures of performance. Here, we follow ideas such as those in Wierman and Harchol-Balter [28] and Raz *et al.* [21] who respectively classify various policies in terms of an unfairness criterion, and propose the resource-allocation queueing fairness measure (RAQFM).

This paper is organized as follows. In Section 2 we describe our general scheduling policy and illustrate its behavior with simulation results. In Section 3 we study several properties of this policy, and analyze the monotonicity of this policy with respect to the parameter, in terms of mean response time and a few kinds of fairness. In Section 4 we present our implementation of this policy in the Linux kernel, discuss a few practical issues and extensions, and demonstrate some experiment results. We conclude in Section 5.

2. THE PBS POLICY

In this section, we describe our configurable, priority-based blind scheduling policy in the context of a single-server queueing system. The system processes *tasks*, which we number sequentially as they arrive to the system. Task i's arrival time is denoted τ_i, where $i < j$ implies that $\tau_i \leq \tau_j$. When $i < j$, we say that task i is *older* than task j, and j is *younger* than i. The total number of arrivals up to time t is denoted by the counting process $\nu(t)$.

The *sojourn time* $t_i(t) = t - \tau_i$ denotes the amount of time that task i has existed in the system by time t. The *attained service time* $x_i(t)$ is the total processor time allocated to task i during time interval $[\tau_i, t]$. The processor is work-conserving with a uniform processing speed, i.e, the total attained service time of all tasks,

$$S(t) := \sum_{i=1}^{\nu(t)} x_i(t),$$

is invariant with respect to the scheduling policy, and increases with a slope of one when the system is busy and zero otherwise. Since $t_i(t)$ grows linearly in time at rate 1 for a task i, and since $x_i(t)$'s growth rate is bounded above by 1, we have that $x_i(t) \leq t_i(t)$ for all i, t.

Each task has a required service time (or task size) of X_i. The definition of a blind policy precludes the policy from using the values of X_i to make scheduling decisions, i.e., one can think of this constraint in practice as the system not knowing X_i until the task i receives all necessary processing (and departs). Let $\tau_i^d = \inf\{t : x_i(t) = X_i\}$ be the departure time of task i, which is when it finishes all its service and leaves the system. After its departure, the attained service time of a task no longer changes, i.e., $x_i(t) = X_i$ for all $t \geq \tau_i^d$. We say a task i is *active* or *in the system* when it is either waiting for or receiving processing. Note that task i is active during and only during the interval $[\tau_i, \tau_i^d]$.

2.1 PBS

A scheduler's basic function is to decide, at any time t, what active task(s) should be processed by the system at that time. The approach in PBS is to associate a time-varying *priority function* with each active task. The general form of PBS's priority functions are $P_i(t) = g(t_i(t), x_i(t))$, meaning that they depend only on the sojourn time and attained service time of a task at that time t. Note the lack of dependence on X_i ensures that the scheduling policy is indeed blind. In this paper, we focus on the specific priority function:

$$P_i(t) = \frac{t_i(t)}{[x_i(t)]^\alpha}, \tag{1}$$

where α is a tunable (constant) parameter between 0 and $+\infty$.

At time t, the scheduler runs active tasks $j(t)$ with maximal priority value in the system,

$$j(t) = \arg\max_{i \in \mathcal{A}(t)} \frac{t_i(t)}{[x_i(t)]^\alpha},$$

where $\mathcal{A}(t) := \{i : 0 < i \leq \nu(t), \tau_i \leq t \leq \tau_i^d\}$ is the set of tasks in the system. If two or more tasks tie as maximum, an arbitrary task is picked. When we analyze PBS in a continuous setting where time slices are infinitesimal, this prioritization scheme translates to a processor that instantaneously shares its processor among all tasks with highest priority. Note, however, that the share is not necessarily equal as will be discussed later in the paper.

Surprisingly, this simple formulation in terms of sojourn time, attained service time, and α enables PBS to emulate or closely approximate several standard blind scheduling policies:

- *FCFS:* As α approaches 0 from above, PBS becomes less sensitive to attained service time. At $\alpha = 0$, if we assume $0^0 = 1$, then $P_i(t) = t_i(t)$, and the earliest arriving task has highest priority, i.e., first-come-first-serve.
- *LAS:* As α tends to ∞, PBS becomes less sensitive to sojourn time, i.e., $[x_i(t)]^{-\alpha}$ dominates the priority value, and the scheduler selects the task with the smallest attained service time, i.e., least-attained-service.
- *PS:* When $\alpha \sim 1$, the priority value of a task is the ratio of sojourn time to attained service time. This ratio is called *slowdown* [28], whose reciprocal is the processor share aggregated over the task's sojourn time. At $\alpha = 1$, the PBS policy tries to maintain an equal slowdown among all tasks by scheduling the one with least aggregated processor share. In this case, its behavior converges as tasks age to PS, but is not strictly identical, unless all tasks have identical arrival time τ_i. Instead, PBS with $\alpha = 1$ has a desirable property that, along every sample path, tasks are scheduled according to their relative slowdown. This means that younger tasks are given a larger fraction of the processor when they become active, and this fraction quickly diminishes to match that of older tasks.
- *Mixture of the above policies.* By varying α continuously from 0 to $+\infty$, we are able to construct an infinite series of scheduling policies that transition in behavior between

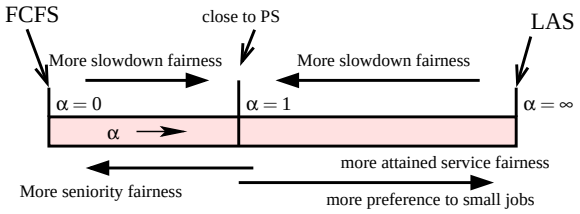


Figure 1: Variation of the PBS policy behavior.

the FCFS, PS and LAS policies; as we increase α, the behavior of the PBS policy changes smoothly: two policies slightly differing α will have similar, but not necessarily identical schedules for a sample set of arrivals.

- *Other well-known blind polices and mixtures.* If we utilize a slightly different priority function of $\text{sgn}(\alpha)t_i(t)/[x_i(t)]^\alpha$, where $\text{sgn}(\alpha)$ is the sign of α, the domain of α can be usefully extended to the entire real set. In other words, for $\alpha < 0$, the scheduler chooses the task with minimum priority $t_i(t)/[x_i(t)]^\alpha$ instead of the maximum. Under this extension, PBS converges to the preemptive last-come first-served (PLCFS) policy as α tends to 0^-, and as α tends to $-\infty$ PBS converges to the LAS policy. Hence, with a decreasing α, we can construct a series of scheduling policies that make smooth transition from the PLCFS policy to the LAS policy. Further discussion of this alternate priority function is beyond the scope of this paper.

Figure 1 provides a high-level overview of the fairness properties one can expect from PBS as a function of α. As α shifts, the policy becomes more fair in terms of one fairness criterion but less fair in terms of another. Hence, the ability to choose α gives the system the ability to decide how to trade off these various types of fairness for its current workload.

An alternate way to view the priority function (which in fact reduces its complexity in an implementation) is to take the logarithm of the original priority function in (1):

$$p_i(t) \equiv \log P_i(t) = \log t_i(t) - \alpha \log x_i(t), \qquad (2)$$

where the base-2 logarithm is used in our implementation.

Since an inequality still holds when the log is taken on both sides, switching from P_i to p_i does not alter the scheduling behavior of PBS. This formulation also reveals that PBS is invariant to the units with which sojourn and attained service times are measured (the scaling of the unit simply factors out as an additive constant, which can be canceled on both sides when comparing two tasks' priority functions).

2.2 Behavior of the PBS policy

Now we illustrate the behavior of the PBS policy with some simulation results and briefly describe and demonstrate how the PBS policy schedules tasks, as shown in Figures 2 and 3. These figures show how the processor is shared among a set of arriving tasks over time for various values of α when the time slices are infinitesimal. In these figures, time increases along the x-axis, with this axis annotated at the top of the figure. A separate simulation is run for each value of α considered, with each multi-colored rectangular box representing a simulation with a fixed value of α. The coloring denotes the fraction of processor each task in the system receives per instant in time (e.g., task 1 always initially receives the entire processor, and later, relinquishes some or all of the processor to task 2). For each value of α, tasks $i = 1, 2, 3, 4$ respectively arrive at times $\tau_i = 0, 1, 3, 5$ and have respective task sizes $X_i = 4.5, 2.5, 3, 2$. One can visually see the change in how the processor is shared, and the change in response times (arrows under the box) as α is varied.

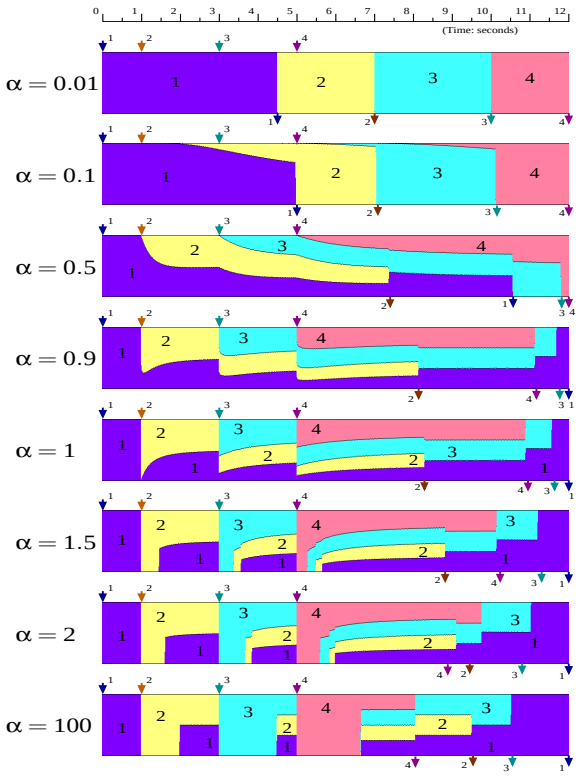


Figure 2: Illustration of processor share that each of four tasks obtains in a 12 second interval.

Before exploring the behavior as a function of α, we wish to point out some general observations:

- By using infinitesimal time-slices, we can view the system as fractionally sharing the processor at any time t. As a system with discrete-sized time slices reduces the size of its slice, an average of the utilization of the processor by tasks within a small, discrete-sized slice will approximate to the fractions seen here.
- Note that for multiple tasks to share the processor at any time t, their priorities must both be maximal, and hence equal. Note also that often these multiple tasks continue sharing the processor, such that their priorities remain equal over time. Finally, note that since in many instances they receive different-sized fractions of the processing, these tasks' priorities are kept even by giving them different fractions of the processor.

For a simple intuition as to how this can happen, consider two tasks 1 and 2 scheduled by a PBS system with $\alpha = 1$. Assume these tasks have different arrival times $\tau_1 < \tau_2$, but have an equal priority at some point in time t. We can write their respective priority functions at this time as $P_1 = t_1/x_1$ and $P_2 = t_2/x_2$, where $P_1 = P_2$. If, during

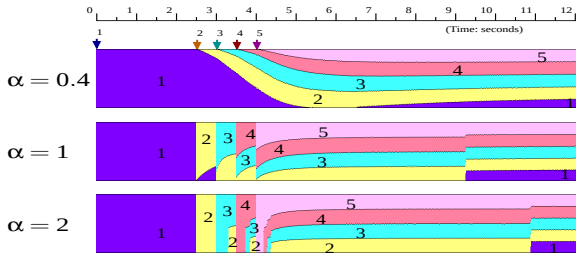


Figure 3: A persistent task sees four new arrivals and gets service interruption.

a short interval of time Δt, both tasks are given half of the processor, their priority functions change value to $(t_1 + \Delta t)/(x_1+0.5\Delta t)$ and $(t_2+\Delta t)/(x_2+0.5\Delta t)$. Note that $P_1 = P_2$ does not imply that the changed values remain equal. Hence, if the priorities are to remain equal over time, the fraction of processor allocated to each task is not necessarily equal.

Let us now informally explore the behavior of PBS as a function of α with Figure 2 guiding the discussion:

- For $\alpha < 1$, an task arriving to the system with a busy processor does not claim the entire processor. For α closer to 0, it gets close to no processing upon initial arrival, but also does not preempt any of its fraction to subsequent arriving tasks (like FCFS). As α moves toward 1, its initial fraction grows, at some point even surpassing the fraction of existing tasks in the system.
- For $\alpha > 1$, a new task immediately occupies the whole processor, preempting all existing tasks. Later on, the task will have to share the processor with the older tasks, and will get preempted by subsequent task arrivals. As α moves downward toward one, the rate at which the new task gives up its fraction of the processor increases. For large α, the behavior is effectively LAS, i.e., the task with the least service thus far gets all the processor, and those tied for least service share the processor equally.
- For $\alpha = 1$, as described earlier, an arriving task initially gets more than $1/N$ of the processor when there are N tasks in the system, but over time, its portion converges to $1/N$, barring adjustments due to new arrivals.

Figure 3 shows arrivals of five persistent tasks ($X_i = \infty$) arriving at times $\tau_i = 0, 2.5, 3, 3.5, 4$ s. What is interesting to note is the share of processor received by the first arriving task. With $\alpha = 0.4$, its share slowly declines to almost nothing and then slowly increases to an equitable share. For $\alpha = 1$, its share drops substantially upon the next arrival, and then to 0 after the third arrival, until a rather large chunk of time passes. For $\alpha = 2$, the first task immediately loses its entire share, getting it back only later. The important observation to make is that in the long-term, as time goes to infinity, all tasks will wind up with an equal share of the processor.

A related parameterized policy is recently proposed by Padhy and Kherani [18, 13], and its discretized version is studied earlier by Kherani and Núñez-Queija [14]. This policy is a discriminated processor-sharing (DPS) policy [9] in which task i's weight equals to x_i^α where x_i is the attained service time and α is the policy parameter. In contrast, the PBS policy uses a priority function (tasks with highest priority values preempt all other tasks), and explicitly considers sojourn time, using the parameter α to trade off favorability towards the arrival time and towards small job sizes. Comparing both policies, it is interesting to see that, although the PBS policy uses priority instead of weights, the involvement of sojourn time results in a time-sharing scheme as a DPS policy can (Figures 2 and 3). On the other hand, the PBS policy may interrupt the service of low-priority tasks, whereas in general all tasks get a share of processor at all time under a DPS policy, except in the extreme cases with zero and/or infinite weights.

The next two sections of this paper can be read in either order, as one does not depend on the results of the other. Section 3 provides further analysis of PBS and its properties for various α, while Section 4 describes our implementation and results demonstrating the practical value of PBS.

3. ANALYSIS OF THE PBS POLICY

In this section, we elaborate more formally on the various properties of PBS. We begin by identifying (and proving when applicable) a number of desirable properties of the policy for interval ranges of α. We then focus on attained service time and mean response time, and explore first and second order behavior of these measures as a function of α and their underlying distribution.

As in much of Section 2, we assume that time slices are infinitesimally small, such that all tasks with the maximal priority at time t share (possibly unevenly) the available processor. We say each task gets an *equal share* if its processor share is $1/N(t)$ where $N(t) := |\mathcal{A}(t)|$ is the number of tasks in the system.

A task in the system is called *scheduled* at time t if it receives a positive processor share *immediately after* t, or in other words, its attained service time is *strictly* increasing at t. More rigorously, it means that this task receives a positive amount of processing time in the time interval $[t, t+\epsilon]$ for any positive ϵ. Note that all scheduled tasks have an identical priority value, which is greater than those of non-scheduled tasks. If a scheduled task becomes non-scheduled after time t, we say that this task gets a *service interruption*.

3.1 Properties of the PBS policy

We now investigate the detailed properties pertaining to the PBS policy. These properties may provide a rough idea on what tasks are scheduled with how much processor share granted, at any time t. Some important properties referred later are stated as theorems with names given.

3.1.1 *Properties for all* $0 \le \alpha < \infty$

Changes in priority with time: A non-scheduled task in the system has a linearly increasing priority value for $\alpha < \infty$. The processor time is (probably unequally) shared by all scheduled tasks whose priority values remain the same as one another, and greater than all other non-scheduled tasks. These priority values can either increase sub-linearly or decrease.

Older gets more (basic fairness): If $i < j$ then $x_i(t) \ge x_j(t)$ for all $t \in [\tau_i, \tau_i^d] \cap [\tau_j, \tau_j^d]$. In other words, at any time such that two tasks are both active, the earlier arriving task has never received less service, as stated in the following theorem.

THEOREM 3.1 (BASIC FAIRNESS). *Under PBS, if $i > k$, $x_i(t) \le x_k(t)$ for any time t such that $i, k \in \mathcal{A}(t)$, i.e., both tasks are in the system.*

PROOF. For $\alpha = 0$ (FCFS), the priority value is the sojourn time. If $i > k$ (task k is older), task i does not get service as long as task k is in the system, i.e., $0 = x_i(t) \le x_k(t)$.

For $\alpha > 0$, we prove this by contradiction. Let us hypothesize that $x_i(t) > x_k(t)$ for some t, and some i, k such that $0 < k < i \le \nu(t)$. Since $x_k(\tau_i) \ge x_i(\tau_i) = 0$ and both $x_i(t)$ and $x_k(t)$ are non-decreasing continuous functions of t, there must be a t such that $x_i(t)$ is strictly increasing (i.e, $x_i(t+\epsilon) > x_i(t)$ for any sufficiently small ϵ). [1] In other words, the task i is scheduled at t, i.e., $t_i(t)/[x_i(t)]^\alpha \ge t_k(t)/[x_k(t)]^\alpha$. However, as a contradiction, for all t such that $x_i(t) > x_k(t)$, we get that $t_i(t)/[x_i(t)]^\alpha < t_k(t)/[x_k(t)]^\alpha$ since $t_i(t) \le t_k(t)$ and $\alpha > 0$. Thus, i has strictly lower priority at time t and would not be scheduled, contradicting the growth of $x_i(t)$ between time t and $t+\epsilon$. □

3.1.2 *Properties for all* $0 < \alpha < \infty$

If we exclude the strict FCFS policy from the spectrum of PBS policies, a number of other beneficial properties hold:

Immediate Service / No infinite priorities: Upon arrival, a new task is immediately scheduled (i.e., every task i gets a positive amount of processing time during interval $[\tau_i, \tau_i+\epsilon)$ for any $\epsilon > 0$,). This can be seen by contradiction: priority can only be ∞ when $x_i(t) = 0$, so if $P_i(\tau_i + \epsilon) = \infty$, then $P_i(\tau_i + \epsilon/2) = \infty$ as well. Because the system is work-conserving and i is active, at time $\tau_i + \epsilon/2$, some task must have received a positive share of processing, and this task's priority would clearly have dropped to a finite value by time $\tau_i 3\epsilon/4$. Hence the number of tasks with infinite priority reduces by 1 before reaching time $\tau_i + \epsilon$. Repeating this argument for all tasks that are active within the interval $\tau_i, \tau_i + \epsilon$, we can find a time before $\tau_i + \epsilon$ where task i is the only task with infinite priority and hence should have been scheduled prior to $\tau_i + \epsilon$. It also follows from this proof that for this range of α, tasks never have infinite priorities after its arrival.

Arrivals (Departures) instantaneously decrease (increase) scheduled tasks' shares: since new arrivals receive immediate service, every scheduled task should experience an instantaneous decrease of processor share after the arrival of a new task, compared to the scenario without this new arrival, because the new arrival receives service immediately, and the processing time is taken from *every* scheduled task. Note that for very small α, the decrease may be very small, but it is always non-zero. Similarly, after a task departs, all other scheduled tasks experience an instantaneous increase of processor share.

Processor share: The processor share at time t of task i, as illustrated in Figures (2) and (3), is defined as $x_i'(t)$, the right-derivative of attained service time at time t. Note that we have $0 \le x_i'(t) \le 1$. This quantity is zero, if a task is not scheduled. If a task is scheduled, the derivative of attained service time can still be zero provided that the attained service time strictly increases at t. If two tasks, i and k, $i < k$, are both scheduled, they should have the same log-priority value:

$$\log t_i(t) - \alpha \log x_i(t) = \log t_k(t) - \alpha \log x_k(t).$$

Taking the derivative we get

$$\frac{x_i'(t)}{x_i(t)} - \frac{x_k'(t)}{x_k(t)} = \frac{1}{\alpha}\left(\frac{1}{t_i(t)} - \frac{1}{t_k(t)}\right), \tag{3}$$

noting that $t_i(t)$ and $t_k(t)$ are linear functions of t. The right-hand side of (3) is positive for $\alpha > 0$ if task i is younger than task k. Then we get

$$\frac{x_i'(t)}{x_i(t)} > \frac{x_k'(t)}{x_k(t)}. \tag{4}$$

Equations (3) and (4) describe the way that the PBS policy allocates processor share to each of the scheduled tasks.

Hospitality: If tasks i and j, $i < j$, are active at time t and $x_i'(t) > 0$, then $x_j'(t) > 0$. In other words, if a task is scheduled, all the active tasks that arrived later than this task also get scheduled. This property is proved by showing that younger tasks always have higher priority than older ones.

THEOREM 3.2 (HOSPITALITY). *Under* PBS *with $\alpha > 0$, if $i > k$, $P_i(t) \ge P_k(t)$ for any time t such that $i, k \in \mathcal{A}(t)$.*

PROOF. Let us hypothesize that $i > k$ and $P_i(t) < P_k(t)$ for some t. By definition of the PBS policy and continuity of both $P_i(t)$ and $P_k(t)$, task i is not scheduled, i.e., $x_i'(t) = 0$, and we get $P_i'(t) = [x_i(t)]^{-\alpha}$. No matter scheduled or not, we have $P_k'(t) = [x_k(t)]^{-\alpha} - \alpha t_k(t)[x_k(t)]^{1-\alpha} x_k'(t) \le [x_k(t)]^{-\alpha}$. By Theorem 3.1(basic fairness) we have $x_i(t) \le x_k(t)$ and from (1) we get that $P_i'(t) \ge P_k'(t)$ for *every* t such that $i, k \in \mathcal{A}(t)$ and $P_i(t) < P_k(t)$.

On the other hand, with $\alpha > 0$, $P_i \ge P_k$ just after its arrival because it receives immediate service. Since $P_i(t)$ and $P_k(t)$ are right-differentiable and continuous, there must be a time t such that $P_i'(t) < P_k'(t)$, based on the hypothesis $P_i(t) < P_k(t)$, using the same argument as in the proof of Theorem 3.1. This causes a contradiction, and hence we always have $P_i(t) \ge P_k(t)$ for time t such that both tasks are in the system. □

Convergence to PS: The PBS policy converges to a PS policy in a long run. Consider several persistent tasks and assume after time t, there are neither new arrivals nor departures. By Theorem 3.2 (hospitality), while an older task is not scheduled, all processing time is given to tasks younger than it. This older task must get scheduled at some time, otherwise sooner or later Theorem 3.1 (basic fairness) will be violated. Therefore, eventually all non-scheduled tasks become scheduled. Then we can look at (3) with two scheduled tasks i, k. The right-hand side of (3) converges to zero as $t_i(t)$ and $t_k(t)$ tends to ∞, and we obtain $x_i'(t)/x_i(t) \approx x_k'(t)/x_k(t)$. Then,

$$\frac{x_i'(t)}{x_k'(t)} \approx \frac{x_i(t)}{x_k(t)} = \left(\frac{t_i(t)}{t_k(t)}\right)^{1/\alpha} \to 1 \quad \text{as } t_i(t), t_k(t) \to \infty,$$

where the second equality is because the priority values of tasks i and k are same, and the limit of one is because $t_i(t) - t_k(t)$ is a constant. This result means that the PBS policy

[1] If $g(\cdot)$ is a continuous function with $g(\tau) \le 0$ and positive at some point later on, we can surely find some $t > \tau$ such that $g(t) > 0$ and $d^+g(t)/dt > 0$ if g is right-differentiable, or more generally $g(t+\epsilon) > g(t) > 0$ for any sufficiently small ϵ. Here $g(t) := x_i(t) - x_k(t)$. We shall use this argument again and again in this paper. If $g(t)$ is differentiable everywhere, this argument corresponds to the mean value theorem in calculus.

will converge to an equal share configuration (i.e., processor sharing), as long as $\alpha > 0$, no matter how small or large the α is. Note that, for α tends to ∞, it is well known that LAS will eventually become PS if there is neither arrival nor departure (for this reason it has another name Foreground-Background Processor-Sharing, FBPS).

No permanent starvation: Here, we consider the starvation problem in a system with a fixed number of users. After having fully received service, each user has non-zero think time before submitting his/her next request of service. Although the system is stable in terms of the number of tasks, there are two kinds of starvation. This first kind of starvation is produced under FCFS: if one user submits a task that fails to end, all subsequent tasks submitted by others are starved. The second kind of starvation is produced under LAS (as well as under MLFS): if two users frequently submit small tasks, a long task submitted by a third user will never get to finish. Under the PS policy, both situations will not result in starvation. Now we consider the PBS policy with $0 < \alpha < \infty$. For the FCFS kind of starvation, we note that no task is starved by a persistent task, as we have shown previously that all tasks will eventually reach an equal share for $\alpha > 0$. For the LAS kind of starvation, since the priority value of a non-scheduled task linearly increases towards infinity and eventually exceeds all short tasks submitted by other users, this task is not permanently starved. However, in order to prevent other users from submitting smaller and smaller tasks to get higher and higher priority, we may impose a lower bound for task sizes on all users. A further discussion is shown in our technical report [10].

Although permanent starvation is avoided, a moderately long term starvation is still possible with very small or large α. Therefore, during implementation, it is necessary to apply a greater-than-zero lower bound and a less-than-infinity upper bound to α. The lower bound is more important because the FCFS kind of starvation is much more common.

3.1.3 Properties for $\alpha \geq 1$

No surprising interruption: If $x_i'(t) > 0$ and no tasks arrive in the interval $[t, t_0]$, then either $x_i(t_0) = X_i$ or $x_i'(t_0) > 0$. In other words, during any period in which no new tasks arrive to the system, all scheduled tasks remain scheduled unless they complete (i.e., they do not become non-scheduled). We justify this property in our technical report [10]. This property need not hold for $0 < \alpha < 1$,as is shown in Figure 3 with $\alpha = 0.4$. (The task #1 becomes non-scheduled at 5 s.)

3.1.4 Properties for $\alpha > 1$

Quick start: $x_i'(\tau_i) = 1$: A newly arriving task takes all the processor at its time of arrival. This is shown by contradiction. Suppose the claim is false and consider time $t = \tau_i + \epsilon$ for a positive small ϵ. Since the task did not get the total processor during this time interval, it received an amount of processing $\delta < \epsilon$. At time t, the priority value of task i is then ϵ/δ^α. We have $\epsilon/\delta^\alpha > \delta/\delta^\alpha = \delta^{1-\alpha}$. Shrinking δ (by shrinking ϵ and maintaining $\delta < \epsilon$), we see that the priority value of task i is unboundedly large (finite though) and can dominate all other existing task priorities. In other words, a new task gets very high priority and remains as the highest for a while after their arrival until they receive enough service, and thus they will interrupt the service of all existing scheduled tasks.

3.1.5 Properties for $\alpha < 1$

Slow start: If there exists task j, $j < i$, that is still active at time τ_i, then $x_i'(\tau_i) = 0$: The proof setup is similar to above. Suppose for every ϵ, the amount of processing received by i by time $\tau_i + \epsilon$ is $0 < \delta \leq \epsilon$. The priority value of task i is $\epsilon/\delta^\alpha \leq \epsilon/\epsilon^\alpha = \epsilon^{1-\alpha}$. Shrinking ϵ, this quantity becomes unboundedly small, meaning that the processor share of a task is zero on its arrival and very small just after its arrival, therefore it could not interrupt the services of other tasks immediately.

3.2 Transient analysis with deterministic model

In Section 3.1, to understand how the policy works, we compare the progress of tasks under PBS with a *fixed* α. This comparison, however, does not give any guideline for tuning the parameter α. In this section and Section 3.3, we compare the PBS policies with different α's and analyze how the performance and fairness change as α changes.

In the beginning of Section 2, we claim the total attained service by all tasks (including all tasks that have departed), namely $S(t)$, is invariant with respect to the scheduling policy, and in particular, invariant with respect to α. Now before getting into our analysis, a few quantities related to $S(t)$ are to be defined. Quantity $S(t,k)$ is the total attained service time by the first k tasks at time t:

$$S(t,k) := \sum_{i=1}^{k} x_i(t), \qquad 0 < k \leq \nu(t).$$

We divide $S(t,k)$ into two portions by a threshold ξ, $0 \leq \xi \leq \infty$, for the attained service time of every task. The first portion is to count only attained service of first ξ seconds of the first k tasks, and the second portion is to count only attained service beyond the first ξ seconds, i.e.,

$$S_\xi^-(t,k) := \sum_{i=1}^{k} [x_i(t) \wedge \xi], \quad S_\xi^+(t,k) := \sum_{i=1}^{k} [x_i(t) - \xi]^+,$$

where $z^+ := z \vee 0$ and we use $\wedge$ and $\vee$ to denote minimum and maximum, respectively. For the case that $k = \nu(t)$, we use shorthand notations

$$S_\xi^+(t) := S_\xi^+(t, \nu(t)) \quad \text{and} \quad S_\xi^-(t) := S_\xi^-(t, \nu(t))$$

as $S(t)$ is in fact a shorthand notation of $S(t, \nu(t))$.

By definition we have $S(t,k) \equiv S_\xi^+(t,k) + S_\xi^-(t,k)$ and $S(t) \equiv S_\xi^+(t) + S_\xi^-(t)$. With $\xi = 0$, we have $S_0^-(t,k) = 0$ and $S_0^+(t,k) = S(t,k)$, whereas, with $\xi = \infty$, we have $S_\infty^+(t,k) = 0$ and $S_\infty^-(t,k) = S(t,k)$.

Now we analyze the PBS policy under two different α's, namely α_1 and α_2 with $\alpha_1 \leq \alpha_2$. The arrival times and sizes of all tasks are assumed to be fixed (i.e., the system is deterministic). We add superscripts to those aforementioned quantities in order to distinguish different policies.

This following theorem states that the total attained service beyond ξ of the first k tasks is greater for a smaller α:

THEOREM 3.3. *Consider two policies* $\mathcal{P}_1 := \text{PBS}(\alpha_1)$ *and* $\mathcal{P}_2 := \text{PBS}(\alpha_2)$. *If* $0 < \alpha_1 \leq \alpha_2 < \infty$, *then* $S_\xi^+(t,k)^{\mathcal{P}_1} \geq S_\xi^+(t,k)^{\mathcal{P}_2}$, *for any* $t \geq 0$, $\xi \in [0, \infty]$ *and* $k = 1, 2, \ldots, \nu(t)$.

See our technical report [10] for a proof of Theorem 3.3.

The monotonicity of the PBS policy with respect to α is probably more clearly implied by the following corollaries of

Theorem 3.3. The first corollary states that the PBS policy assigns more processing time to older tasks at any time if a smaller α is used, meaning that by decreasing α we increase the favorability towards the older tasks.

COROLLARY 3.4. *Consider two policies* $\mathcal{P}_1 := \text{PBS}(\alpha_1)$ *and* $\mathcal{P}_2 := \text{PBS}(\alpha_2)$. *If* $0 < \alpha_1 \le \alpha_2 < \infty$, *then* $S(t,k)^{\mathcal{P}_1} \ge S(t,k)^{\mathcal{P}_2}$, *for any* $t \ge 0$ *and* $k = 1, 2, \ldots, \nu(t)$.

PROOF. Use Theorem 3.3 and set ξ to zero. □

The second corollary states that the PBS policy assigns more service to small tasks at any time if a larger α is used, meaning that by increasing α we increase the favorability towards the small tasks.

COROLLARY 3.5. *Consider two policies* $\mathcal{P}_1 := \text{PBS}(\alpha_1)$ *and* $\mathcal{P}_2 := \text{PBS}(\alpha_2)$. *If* $0 < \alpha_1 \le \alpha_2 < \infty$, *then* $S_\xi^-(t)^{\mathcal{P}_1} \le S_\xi^-(t)^{\mathcal{P}_2}$, *for any* $t \ge 0$ *and* $\xi \in [0, \infty]$.

PROOF. Use Theorem 3.3 with $k = \nu(t)$, we get $S_\xi^+(t)^{\mathcal{P}_1} \ge S_\xi^+(t)^{\mathcal{P}_2}$. The corollary follows since $S(t) = S_\xi^+(t)^{\mathcal{P}_1} + S_\xi^-(t)^{\mathcal{P}_1}$ is policy invariant. □

The third corollary states that the variance of attained service time is smaller if α is larger.

COROLLARY 3.6. *Consider two policies* $\mathcal{P}_1 := \text{PBS}(\alpha_1)$ *and* $\mathcal{P}_2 := \text{PBS}(\alpha_2)$. *Define*

$$\sigma(t) = \frac{1}{\nu(t)} \sum_{i=1}^{\nu(t)} \left[x_i(t) - \frac{S(t)}{\nu(t)} \right]^2.$$

If $0 < \alpha_1 \le \alpha_2 < \infty$, *then* $\sigma(t)^{\mathcal{P}_1} \ge \sigma(t)^{\mathcal{P}_2}$, *for any* $t \ge 0$.

PROOF. Since $\nu(t)$ and $S(t)$ are policy invariant, it is sufficient to prove just for the second moment, i.e.,

$$\sum_{i=1}^{\nu(t)} \left[x_i(t)^{\mathcal{P}_1} \right]^2 \ge \sum_{i=1}^{\nu(t)} \left[x_i(t)^{\mathcal{P}_2} \right]^2$$

holds. In fact, the preceding inequality follows Theorem (3.3) due to

$$\sum_{i=1}^{\nu(t)} [x_i(t)]^2 = \sum_{i=1}^{\nu(t)} \int_0^{x_i(t)} 2[x_i(t) - \xi] d\xi$$
$$= 2 \int_0^\infty \left[\sum_{i=1}^{\nu(t)} [x_i(t) - \xi]^+ \right] d\xi = 2 \int_0^\infty S_\xi^+(t) d\xi.$$

Then the corollary immediately follows. □

Remarks on fairness. Aside from traditional performance measures such as the response time and throughput, another concern when designing a scheduling policy is the fairness of the system. This refers roughly to the individual experience of a particular task with respect to its peer tasks. As this "experience" equals out across the various types of tasks, the system is deemed to be more "fair". Different fairness criteria have been proposed and analyzed in the recent past. Possible criteria for comparing scheduling policies are (in our own terminology): *share fairness* (the proportion of processing time allocated to a task at any moment), *seniority fairness* [21] (the time spent in the system), and *slowdown fairness* [28] (the ratio of response time to task size). Clearly, the PS policy attains the most share fairness, and the FCFS policy gets the most seniority fairness.

From the corollaries in this section, we can see that the PBS policy is monotonic in terms of some fairness measures: Corollary 3.4 implies the seniority fairness gets monotonically improved as α decreases; Corollary 3.6 implies that the variance of the attained service time monotonically decreases as α increases, which can be also considered as a measure of fairness. We refer to it as *attained service fairness*. For slowdown fairness, the PBS policy at $\alpha = 1$, however, tries to attain the highest slowdown fairness at every moment. (It always schedules tasks that have the maximum slowdown so as to decrease it.) Therefore, when α varies between zero and infinity, the PBS policy is actually doing a trade-off between seniority fairness, attained service fairness, and slowdown fairness (as shown in Figure 1).

We defined the "basic fairness" in Section 3.1: the PBS policy maintains this property that at any time, an older task is always running ahead of a younger one, as stated by Theorem 3.1. As each of the PS, FCFS, and LAS policies satisfies this property, it is not seen in some other scheduling policies: the PLCFS policy is just on the opposite extreme; both the Shortest Job First (SJF) and the Shortest Remaining Processing Time (SRPT) policies [26] are certainly far from compliance. This basic fairness is seemingly contrary to the "hospitality" property that the younger tasks always get assigned, as implied by Theorem 3.2. These two properties and the three corollaries in this section constitutes the full fairness spectrum of the PBS policy.

The response time of the tasks, however, have not been addressed. In fact, there is no way to improve the response time of every task: due to work-conserving principle, favoring small tasks causes a penalty to large tasks, and speeding up older tasks causes a delay to younger tasks. What we can do is to improve the response time on average. A stochastic model is therefore needed for further analysis.

3.3 Stationary analysis with stochastic model

In this section, we assume the task sizes $X_1, X_2, \cdots$ are independent, identically distributed random variables, with the same distribution as the random variable X with $\mathbb{E}X = 1/\mu$. The cumulative distribution function and the probability density function of X are denoted by $F(\cdot)$ and $f(\cdot)$, respectively. We also assume that the arrival process $\nu(t)$ is a stable counting process with a fixed rate of λ. In other words, we assume a $G/GI/1$ work-conserving queue.

The response time $T_i := \tau_i^d - \tau_d$ of task i is defined to be its sojourn time on its departure. Dropping the indices, the response time is denoted by the random variable T. We are also interested in the response time conditioned on the task size x, denoted by T_x. The mean conditioned response time $\mathbb{E}T_x := \mathbb{E}[T|X = x]$ satisfies that $\mathbb{E}T = \int_0^\infty \mathbb{E}T_x dF(x)$. For stable queues, the mean response time is equal to the time average: $\mathbb{E}T = \lim_{t\to\infty} \frac{1}{\nu(t)} \sum_{i=1}^{\nu(t)} T_i$. It is well known that the SRPT policy minimizes $\mathbb{E}T$ [25], but SRPT is not blind and is therefore not applicable to many systems. For a blind policy, unfortunately, the Kleinrock's conservation law [15] states that the quantity $K := \int_0^\infty \mathbb{E}T_x[1 - F(x)]dx$ is invariant with respect to the policy, and therefore the mean response time for exponential distribution is also policy invariant, since in this case $1 - F(x) \equiv f(x)/\mu = e^{-\mu x}$ and hence mean response time is proportional to Kleinrock's constant K. Nevertheless, in many systems, the task size is far from

exponentially distributed. Evidently the task sizes exhibit a heavy-tailed property in a great portion of the current computer and network systems [5]. The class of distributions with decreasing (increasing) hazard rate, DHR(IHR) is sometimes used to characterize the heavy-tailed(light-tailed) task sizes [1]: a probability distribution is DHR (IHR) if the hazard rate $f(x)/1-F(x)$ is decreasing (increasing).

The following theorem states that for these two classes of task-size distributions, the mean response time is monotonic with respect to the policy parameter α.

THEOREM 3.7. *Consider two policies* $\mathcal{P}_1 := \text{PBS}(\alpha_1)$ *and* $\mathcal{P}_2 := \text{PBS}(\alpha_2)$. *If* $0 < \alpha_1 \leq \alpha_2 < \infty$, *then in a* $G/GI/1$ *queue,* $\mathbb{E}T^{\mathcal{P}_1} \geq \mathbb{E}T^{\mathcal{P}_2}$ *with DHR task-size distributions, and* $\mathbb{E}T^{\mathcal{P}_1} \leq \mathbb{E}T^{\mathcal{P}_2}$ *with IHR task-size distributions.*

See our technical report [10] for a proof of Theorem 3.7. With $\alpha = \infty$ ($\alpha = 0$), we can see that Theorem 3.7 is consistent with the known fact that LAS (FCFS) is the optimal blind policy for DHR (IHR) task-size distributions in terms of mean response time [22].

According to Theorem 3.7, we should increase the α value in order to reduce the mean response time with a heavy-tailed task-size distribution, and the best choice is $\alpha = \infty$ (LAS). However, as discussed earlier, the LAS policy has starvation problem, so it might be desirable to have a finite upper bound for α. Furthermore, the mean response time is not the only criteria that we need to consider; using a α that is too large may result in unacceptably bad seniority and slowdown fairness.

3.4 Comparison to MLFS/MLPS

The multi-level processor sharing (MLPS) [15] (or multi-level feedback-queue scheduling, MLFS, as referred to in operating system implementations [27]) is a blind policy that divides the attained service time into multiple levels. Each level contains an interval of attained service time, and tasks in the level of the shorter attained service time has absolute priority over those in the level of longer attained service time. For tasks in the same level, any blind scheduling policy can be used, e.g., LAS, FCFS, or PS.

With many levels, the MLPS policy is just an approximation of LAS policy and the scheduling policy within each level becomes less important. Therefore, a MLPS policy with two levels has received a lot of attention recently [1], in which attained service time is divided by a single threshold, with the first level using LAS and the second level using PS. In fact, using a tunable threshold between two levels, we can make this two-level MLPS policy configurable, representing a series of mixed policies between LAS and PS.

Comparing with PBS, we can see a few downsides in the tunable two-level MLPS. First, unlike PBS, using the threshold as tunable parameter in MLPS, the behavior depends on the unit of the attained service time. Second, between levels, it still functions like an LAS policy, therefore the first-level tasks can starve the second-level ones. Third, the sojourn time information is never used, and thus there is no way to consider the seniority fairness. It can neither emulate FCFS nor be anywhere close, and the mean response time is not good in case that the task sizes have small variability. Last, but not the least, tasks experience a sudden slowdown after it is demoted to the second level. The upside of the two-level MLPS are that it may be more analytically tractable.

4. IMPLEMENTATION AND RESULTS

This section presents the implementation of the PBS policy into Linux kernel version 2.6.15.7, and results from experiments using the implementation. We identify the value of α for which PBS closely emulates the original Linux scheduler, and then demonstrate that for different workloads, the value for α that minimizes expected waiting time is different, justifying the practical benefits of having a scheduler whose behavior is easily tuned.

Kernel version 2.6.15.7, within which we implemented PBS, was the most recent stable version of the Linux kernel. Our implementation was limited to single processor machines and hence the kernel was configured to single processor (i.e., not SMP) mode. Dividing the current count of clock cycles obtained through the Pentium instruction `rdtsc` by the CPU frequency, we obtained accurate measurements of sojourn time and attained service time of each process, excluding the time spent by the scheduler and the interrupt handler. We implemented a 64-bit integer logarithm in order to compute the log-priority value. Parameter α is stored in the kernel as a variable, and a new system call `lips_ctrl(int cmd, int pid, void* param)` is added to the kernel. This call is accessible from the user level and allows user-level code to set or get the current value of α. This system call is also used to get scheduler statistics, and to set and get other per-cpu or per-process parameters. We shall discuss some of these parameters in this section.

4.1 Practical Challenges

Transitioning from a continuous description of the scheduler to a discretized description that can be implemented in practice required us to make several design decisions. In this subsection, we explain the issues, our decision, and justification.

Discretized Timeslice. PBS is described previously in the context of a system that can divide time into infinitesimal timeslices. This is unrealistic in actual systems, as the timeslice must be some positive, contiguous chunk of time. We stick with the Linux default setting of 4 ms.

Handling of blocked tasks. Processes can remain alive yet be *blocked*, meaning that it cannot utilize the processor because some resource it needs (e.g., disk, I/O devices) is unavailable. One way to handle such blocking would be to simply ignore it, i.e., t_i grows continually at all times, and x_i increases when a task is not blocked and receives the processor. To see the problem with this approach, consider two tasks, i and j where i arrives to the system long before j ($t_i \gg t_j$) and i is blocked for a long time such that $x_i \ll x_j$. When i is no longer blocked, its larger t_i and smaller x_i will enable it to starve out task j, regardless of the value of α. Some services, such as http requests in Apache servers can block for hours, days, or months, and will starve the rest of the system from a long time upon its activation.

To avoid this type of task starvation, we can freeze t_i for a task that is blocked. This effectively freezes the task's priority level, which can be viewed as unfair to the blocked task since the priority of unblocked tasks that are not being processed continue to grow. Instead, our approach to avoid for this unfairness is to reset both t_i and x_i to 0 for a task whose blocked period is an excessive amount of time. We measure this time with a parameter β indicating the multiple of the average blocking time over which we reset x_i and t_i. In our experiments, we set $\beta = 3$, which means once a task

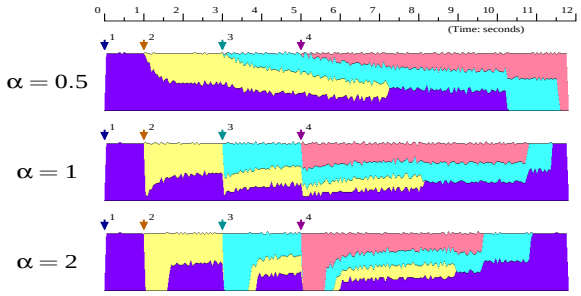


Figure 4: Processor share on the modified Linux.

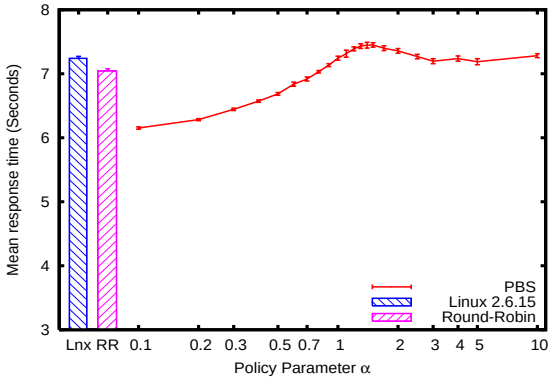


Figure 5: The mean response time for the computational model (Scenario 1).

is blocked for more than three times the average blocking period, its priority is set to that of a newly arriving task.

4.2 Sanity Check of Linux PBS

In our first set of experiments, we utilized the same scenario considered in Section 2, with four independent processes starting at 0 s, 1 s, 3 s, and 5 s, and running for roughly 4.5 s, 2.5 s, 3 s, and 2 s. However, in this section, the processes are run within an actual Linux implementation as opposed to on a simulator that runs an idealized (continuous) implementation of PBS. During its running time, each process repeatedly invokes a subroutine that takes a constant running time, and records the time between invocations by checking the system time. Figure 4 displays the dynamics of the processor share obtained by each process over time, averaged over a 50 ms window. Comparing Figure 4 with Figure 2, we can see that, for each value of α considered, the modified scheduler in the Linux kernel implementation's partitioning of processor among tasks has some additional variance due to the unavoidable discretization of the timeslices, but whose basic partitioning is identical.

4.3 Finding the best α

We now turn our attention to identifying good values of α. We demonstrate that, even when the only objective of interest is the mean response time, the best value of α depends on the distribution of task sizes.

To reduce the interference from other running services and to ensure a fair comparison, we use a self-compiled Linux From Scratch version 5.1.1 Linux distribution [6] with our modified kernel in all experiments.

4.3.1 Scenario 1: Small α best

We consider a setting in which eight users perform computations simultaneously. Each computation consists of a sequence of simple mathematical operations, lasting approximately three seconds, and is divided into six segments. Between segments, each user experiences a short blocked period (e.g., in order to wait for the data from the disk), which lasts about 10 ms. Between computational tasks, each user has a random think time, exponentially distributed with mean 25 s.

In each experiment, we measured the mean response time of all tasks for 7200 s, with 800 s ramp-up time and 200 s ramp-down time. For each α, we repeated experiments for 20 times on a Pentium III 550 MHz desktop computer. Figure 5 plots the mean response time under the PBS policy as a function of α. For each α, we show in Figure 5 the average and 99% confidence interval according to the 20 samples of mean response time. For comparison, the bars in Figure 5 shows the results from similar experiments using either the Linux 2.6.15 native scheduler or a standard RR (round-robin) one.

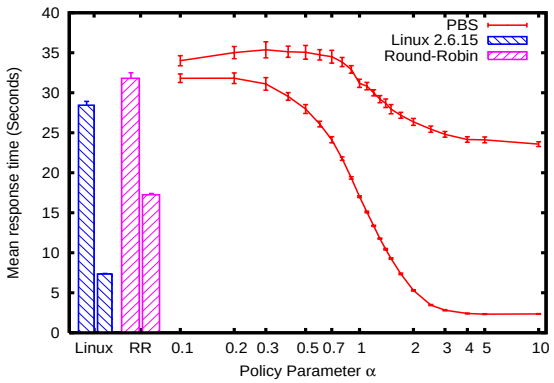
The results show that the mean response time under the RR scheduler is slightly lower than that under the native scheduler of Linux 2.6.15. On the other hand, compared to the native scheduler, the excess of mean response time (the amount beyond the required running time of 3 seconds) is reduced by 20-30% under the PBS policy with small values of α (0.1–0.4). We see that, in such a system with tasks of similar sizes, smaller values of α are better.

4.3.2 Scenario 2: Large α best

Our other scenario utilizes an Apache httpd server [3] (2.0.55) on the same hardware as above. A total number of 30 clients generate HTTP requests with random think time exponentially distributed with 10 s mean. The requested file sizes are described by the Pareto distribution with a shape index of 1.2 [12]. Dynamic web pages are generated by a CGI program. This program reads random file contents from disk with additional processing to consume CPU time, and returns a short summary page to the client in order to save network bandwidth. In general, our experiments run with a CPU-bound system instead of a network-bound one. The processing time is proportional to the file size requested by the clients.

Figure 6 shows the mean response time for different α's, compared with Linux and RR schedulers. The figure uses the same format as Figure 5. However, for each type of scheduler, we include an additional plot whose values lie beneath the original to identify the mean response time of "small" tasks, where we define a task as "small" if its size lies below the mean size taken over all tasks in the experiment. Roughly 78% of the requests are "small." The results show that, for web-like workload, the native Linux scheduler yields a smaller mean response time in comparison to the RR scheduler; for small tasks the gain is more significant. Again, we see that PBS can achieve an even lower mean response time than both the both Linux and RR schedulers, but in this scenario, α must be large. The benefit of using PBS with a large α is considerably more dramatic when we consider the mean response time of small tasks.

4.4 Parameter Extensions



i

Figure 6: The mean response time for the web server model (Scenario 2).

During the development and testing of our implementation, we discovered some minor limitations of the simple, single-parameter priority mechanism described thus far. We conclude this section by exploring these limitations and showing how, by adding additional parameters, PBS can address these limitations.

4.4.1 Context switch cost

In real systems, when the processor context switches (suspends processing a task i and resumes or starts processing another task j), there is a brief period of time during which no task is processed, representing a cost imposed by context switches. For interactive tasks that require a high level of responsiveness, frequent switching is necessary, and the expected time between context switches should be kept short. In contrast, long-running tasks that do not require a high level of responsiveness are better served with longer times between context switches.

Regardless of the value of α, there are occasions where PBS will context switch after every time slice (i.e., every 4 ms using the default Linux setting). For example, when two tasks arrive at approximately the same time, their priority values vary similarly. In each time slice, the task with slightly higher priority will be processed, and the processing in this interval will lower its priority against the other task, inducing a context switch afterwards.

The PBS policy can be easily and elegantly extended to increase expected times between context switches by including an additional parameter, γ. In an actual implementation, in any given time interval, only a single task i is processed at a time. When comparing priority values of tasks to determine the task for processing in the next time-slice, we can prioritize the current processed task i by requiring other tasks' priorities to exceed task i's priority by at least γ, i.e., task j is not swapped in for task i unless $p_j(t) > p_i(t) + \gamma$.

We emphasize two points about including this additional parameter:

- The value of γ does not affect a task's share of processor over long periods of time, but simply reorders on shorter timescales the slices during which it receives the service.
- As tasks age, the expected time between context switches increases.

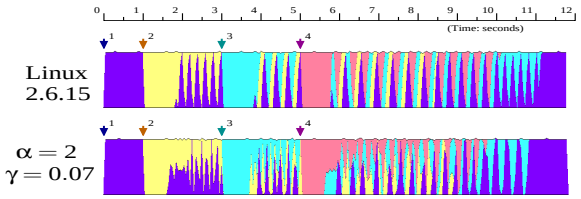


Figure 7: Comparison of the native Linux scheduler and the PBS scheduler. The processor share of each process is the average over a 50 ms window.

To see a rough sketch of the second claim above, consider that n persistent tasks have been running for a fairly long period of $t_i(t) \approx \tilde{t}$ seconds, and the scheduler grants each task i a share that is roughly equal, i.e.,. $x_i(t) \approx \tilde{x} = \tilde{t}/n$. Suppose the scheduler switches to a task at time $\tilde{t}$. We add a bonus γ to the log-priority value $p_i(t)$ (as shown in (2)) of this task. After Δt seconds, the log-priority value of the current task is approximately

$$\log(\tilde{t} + \Delta t) - \alpha \log(\tilde{x} + \Delta t)) + \gamma \tag{5}$$

and that of a non-scheduled other task is

$$\log(\tilde{t} + \Delta t) - \alpha \log \tilde{x} \tag{6}$$

When (6) exceeds (5), there is a context switch. The elapsed time between two consecutive context switches is then

$$\Delta t \approx \tilde{x} \left(2^{\gamma/\alpha} - 1\right) = \frac{\tilde{t}}{n} \left(2^{\gamma/\alpha} - 1\right).$$

We can see that the time between the context switch is indeed increasing and approximately linear to the attained service time in a long run.

We can also use this intuitive derivation above to estimate a good value for γ, assuming that the desired expected switching time, Δt, is known and is a function that is inversely proportional to the attained service time, $x_i(t)$:

$$\gamma \approx \alpha \log\left(1 + \frac{n\Delta t}{\tilde{t}}\right) \approx \alpha \log\left(1 + \frac{\Delta t}{\tilde{x}}\right). \tag{7}$$

For example, suppose we wish to configure PBS with a value of $\alpha = 2.0$ such that $\Delta t/x_i(t) = 1/40$, meaning that tasks are expected to run continuously for periods roughly equal to 1/40 of their attained service time (e.g., the time between context switches is 100 ms when the task has been running for 4 seconds). In this case, we choose γ to be

$$\gamma \approx \alpha \log(1 + 0.025) \approx 0.07.$$

Note that this value is very small. An experimental comparison of the scheduler with and without this bonus is given in Figure 7. The figure shows that, with $\alpha = 2$ and $\gamma = 0.07$, the behavior of the PBS policy is very similar to that of the Linux 2.6.15 native scheduler for the first few seconds, except that with PBS the context switch frequency decreases over time. [2]

4.4.2 Weighted priorities (nice)

[2]We found that the behavior of a different Linux 2.6 version is in fact different; for example, the scheduler in later Linux versions (e.g. 2.6.20) is more close to the RR policy.

Many operating systems provide applications a chance to change their priority within the system. In Unix-like systems, the priority shift is implemented by the system call `nice`. However, a poorly implemented `nice` will starve lower priority tasks. We can extend PBS with an offset parameter, δ_i, per task (δ_i can be either positive or negative) to the calculated log-priority value for task i. No permanent starvation will be introduced (as shown in Section 3.1), but one should be careful not to make δ_i values too large. To make task i's long-term share of processor a multiple κ_i of the share of a regular task k (where $\delta_k = 0$), we set $x_i(t) \approx \kappa_i x_k(t)$ and $t_i(t) \approx t_k(t)$ in

$$\delta_i + \log t_i(t) - \alpha \log x_i(t) = \log t_k(t) - \alpha \log x_k(t)$$

yielding

$$\delta_i \approx \alpha \log \kappa_i, \tag{8}$$

or equivalently $\kappa_i = \exp(\delta_i/\alpha)$. With δ_i given to each task, the PBS policy converges to to a DPS policy [9].

5. CONCLUSION

We have designed, analyzed, implemented, and evaluated a generalized priority-based blind scheduling policy, called PBS. By appropriately setting the scheduler's tunable parameter, the scheduler is able to approximate well-known blind policies like FCFS, PS, and LAS policies, as well as implement policies that trade off the various desirable fairness properties of the well-known policies. We also demonstrated analytically and empirically that for different workloads, system performance is optimized under different settings of the parameter. Hence, systems that must utilize blind policies to schedule their tasks can easily handle a wider variety of workloads with such a tunable scheduler. While we have guidelines at this time to give insight how to tune the parameter, we don't have closed form results on optimality. One direction for future work is to provide better mechanisms for tuning this parameter, one possibility being a closed-loop controller that utilizes feedback from the system on its current performance to make on-line adjustments to the parameter.

6. REFERENCES

[1] S. Aalto, U. Ayesta, and E. Nyberg-Oksanen. Two-level processor-sharing scheduling disciplines: Mean delay analysis. In *Proc. ACM SIGMETRICS '04*, pages 97–105, 2004.

[2] J. Abate and W. Whitt. Limits and approximations for the $M/G/1$ LIFO waiting-time distribution. *Operations Research Letters*, 20:199–206, 1997.

[3] Apache Software Foundation. Apache Httpd. http://httpd.apache.org/.

[4] N. Bansal. Achievable sojourn times by non-size based policies in a GI/GI/1 queue. Technical report, IBM Watson Research Center, 2004. http://www.research.ibm.com/people/n/nikhil/papers/blindnew.pdf.

[5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. SIGMETRICS/PERFORMANCE '98*, pages 151–160, November 1998.

[6] G. Beekmans. Linux from scratch. http://www.linuxfromscratch.org/lfs/.

[7] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng. Group ratio round-robin: $O(1)$ proportional share scheduling for uniprocessor and multiprocessor systems. In *Proc. USENIX '05*, 2005.

[8] E. Coffman, R. Muntz, and H. Trotter. Waiting time distribution for processor-sharing systems. *Journal of the ACM*, 17(1):123–130, 1970.

[9] G. Fayolle, I. Mitrani, and R. Iasnogorodski. Sharing a processor among many job classes. *Journal of the ACM*, 27(3):519–532, 1980.

[10] H. Feng, V. Misra, and D. Rubenstein. The PBS policy: Some properties and their proofs. Technical Report CUCS-015-07, Dept. of Computer Science, Columbia University, 2007.

[11] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. SOSP*, 2003.

[12] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2), 2003.

[13] A. Kherani. Sojourn times in (discrete) time shared systems and their continuous time limits. In *Proc. Valuetools '06*, 2006.

[14] A. Kherani and R. Nùñez-Queija. TCP as an implementation of age-based scheduling: Fairness and performance. In *Proc. IEEE Infocom '06*, 2006.

[15] L. Kleinrock. *Queueing Systems Volume I: Theory, Volume II: Computer Applications*. John Wiley&Sons, 1975,1976.

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[17] I. Mitrani and J. H. Hine. Complete parameterized families of job scheduling strategies. *Acta Informatica*, 8:61–73, 1977.

[18] S. Padhy and A. A. Kherani. Tail equivalence for some time-shared systems. In *Proc. Valuetools '06*, 2006.

[19] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[20] I. A. Rai, G. Urvoy-Keller, M. K. Vernon, and E. W. Biersack. Performance analysis of LAS-based scheduling disciplines in a packet switched network. In *Proc. ACM SIGMETRICS '04*, pages 106–117, 2004.

[21] D. Raz, H. Levy, and B. Avi-Itzhak. A resource-allocation queueing fairness measure. In *Proc. ACM SIGMETRICS '04*, pages 130–141, 2004.

[22] R. Righter, J. G. Shanthikumar, and G. Yamazaki. On extremal service disciplines in single-stage queueing systems. *Journal of Applied Probability*, (2):409–416, 1990.

[23] M. Ruschitzka and R. Fabry. A unified approach to scheduling. *Commun. of the ACM*, 20(7):469–477, 1977.

[24] L. Schrage. The queue $M/G/1$ with feedback to lower priority queues. *Management Science*, 13(7):466–474, 1967.

[25] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[26] L. E. Schrage and L. W. Miller. The queue $M/G/1$ with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1966.

[27] A. Silberschatz, P. B. Galvin, and G. Gagne. *Applied Operating Systems Concepts*. John Wiley&Sons, 2000.

[28] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an $M/GI/1$. In *Proc. ACM SIGMETRICS '03*, pages 238–249, 2003.

[29] A. Wierman, M. Harchol-Balter, and T. Osogami. Nearly insensitive bounds on smart scheduling. In *Proc. ACM SIGMETRICS '05*, pages 205–216, 2005.