PBS: A Unified Priority-Based CPU Scheduler

Hanhua Feng
Computer Science
Columbia University

Vishal Misra
Computer Science
Columbia University

Dan Rubenstein
Electrical Engineering
Columbia University

May 1, 2006

Abstract – A novel CPU scheduling policy is designed and implemented. It is a configurable policy in the sense that a tunable parameter is provided to change its behavior. With different settings of the parameter, this policy can emulate the first-come first-serve, the processing sharing, or the feedback policies, as well as different levels of their mixtures. This policy is implemented in the Linux kernel as a replacement of the default scheduler. The drastic changes of behaviors as the parameter changes are analyzed and simulated. Its performance is measured with the real systems by the workload generators and benchmarks.

1 Introduction

The performance of a system that must simultaneously process multiple tasks depends not only on the caliber of the hardware system, but also on the scheduling policy, which decides how and when the various tasks waiting for service are processed. Although the scheduling algorithms for the operating systems have been studied for many years, emerging demands result in significant changes in the scheduling code of operating systems, and stimulate the resurgence of the research [13]. These demands arise in many different areas, for example, change of workload patterns, and new multi-processor architectures [10]. One can observe their immediate impact on operating system products like Linux [1] whose scheduling code is constantly changing over recent years.

In computer and network systems, scheduling policies usually use only past information, for example the arrival time and received processing time of a task, to decide the current allocation of processing resources, because operating systems hardly know the processing requirements of tasks beforehand, and networks transport flows whose sizes are not notified in advance. This class of scheduling policies is called blind [5]. Some common blind policies are First-Come First-Served (FCFS) which uses only a task's arrival time, Processor Sharing (PS) which splits processing time equally across all current tasks, and Last 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 [21]. The performance of these blind scheduling policies is not only theoretically well-studied [15] but also widely implemented in operating systems. Although the round-robin (RR) policy with fixed time slices, as an implementation of PS, is the *de facto* standard for the schedulers in computer and network systems, the multi-level feedback scheduling [23] is found in many operating systems for elevating as of interactive tasks and boosting overall performance. As RR and PS policies are close approximations of each other, the multi-level feedback algorithm can be understood as an implementation and approximation of the LAS policy, with a finite number of priority levels [15]. Various modifications to these standard policies have been introduced; for example, the Linux scheduler gives a priority bonus to interactive tasks, where the interactiveness level of a task is determined by the processing time it consumed in the past versus the time since its creation.

Nevertheless, none of these standard and modified scheduling policies can be claimed as the best. There are several reasons why it is not possible to develop a universal policy that fits in all circumstances:

- *Workload characteristics change.* The workload of an operating system depends on its applications. For example, the workload for a dedicated Linux firewall server is drastically different from that of a computational server. Even with a fixed installation, the workload changes over time. One can experience its change after a user decides to install a P2P file-sharing software package.

- *Performance needs vary.* For example, the same operating system may be either used in a desktop, whose responsiveness is paramount, or in a server, whose high throughput is anticipated. For real-time systems, the performance may be evaluated by the percentage of real-time tasks that finish before their deadlines [16].

- *Hardware infrastructures differ.* For example, a user is likely to complain the unresponsiveness of an aged computer after upgrading the operating system.

Certainly capable software developers can implement multiple schedulers in the operating system kernel. Despite its viability, the system becomes more complicated: separated pieces of code introduces uneven code quality, and the maintenance needs much more efforts. A better alternate is to use a *configurable policy*, a single policy whose formulation is well established, whose behavior can be fine tuned with parameters, whose implementation appears in a single piece of code.

In this paper, we present such a policy, which we name Priority-based Blind Scheduling or PBS policy. This is a configurable policy that unifies many common blind scheduling policies under a single umbrella. It contains a single tunable parameter that can be set to specific values so as to closely approximate well-known blind policies. The parameter can also be adjusted continuously, permitting a smooth transition between those well-known blind scheduling policies, enabling hybrid policies that mix the various benefits of the well-known policies between which they lie. Briefly, the contributions of our paper are:

- a unified treatment of a wide variety of well known blind scheduling polices, leading to configurable policies that provide balance trade-offs between performance and fairness,

- a novel scheduler that approximates well-known blind scheduling policies, as well as the mixtures of them through a tunable parameter,

- analysis and simulations to discover and illustrate the behaviors of this configurable policy,

- an implementation of this policy as an alternative scheduler of current Linux systems, and

- experimental measurement results to demonstrate the usability and feasibility of this policy.

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 also identify various theoretical properties of PBS that hold under various ranges of α, and investigate some practical challenges of implementing the "theoretical" PBS policy in a real environment.

This paper is organized as follows. In Section 2 we describe our general scheduling policy. In Section 3 we study the transient behavior and properties of this policy, and give some simulation results. In Section 4 we present our implementation of this policy in the Linux kernel and some experiment results. In Section 5, we discuss a few practical issues and extensions. We conclude in Section 6. Some mathematical deductions are provided in Appendix.

2 The PBS Policy

Our new scheduling policy maintains a minimum set of past information, which includes the *sojourn time* $T_i(t)$ and the *processing time* $S_i(t)$ (or *attained service time*) for task i. The sojourn time $T_i(t)$ is $t - \tau_i$, where t is the current clock time and τ_i is the task's *arrival time*. The inequality $S_i(t) \leq T_i(t)$ always holds.

The priority value of task i is given by function $P_i(t) = g(T_i(t), S_i(t))$. In this paper, we use a priority function of

$$g(T, S) := \frac{T}{S^\alpha}, \tag{1}$$

where α is a configurable parameter between 0 and $+\infty$. At clock time t, the scheduler runs task $j(t)$ that has a maximum priority value:

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

where $\mathcal{A}$ is the set of tasks that can be scheduled. Since this policy does not use, or try to predict, any future information, we refer to it as a Priority-based Blind Scheduling (PBS) policy.

With this α, the PBS policy emulates many popular scheduling policies:

- *The FCFS policy*. If α equals to 0, because $S^0 \equiv 1$ (we define $0^0 = 1$ as well), the scheduler simply chooses the one with the greatest sojourn time, or equivalently, the one with the earliest arrival time. Therefore, it is identical to the first-come first-served (FCFS) policy.

- *The LAS policy*. As α tends to ∞, the PBS policy is equivalent to the least attained service (LAS) policy because, for an astronomically large α, $S_i^\alpha(t)$ becomes a dominant factor in calculating the priority value, and the scheduler will choose the one with the least attained service time.

- *The PS policy*. If α equals to 1, the priority value of a task is the ratio of sojourn time to processing time. This ratio is called *slowdown* [26], 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 is similar, but not identical, to that of the PS policy. (It is identical to the PS policy only if the arrival times of all tasks are same.)

- *Mixture of the above policies*. If we change α from 0 to $+\infty$, we are able to construct a series of scheduling policies that are somewhere between the FCFS policy and the LAS policy; as we increase α, the behavior of the PBS policy changes smoothly.

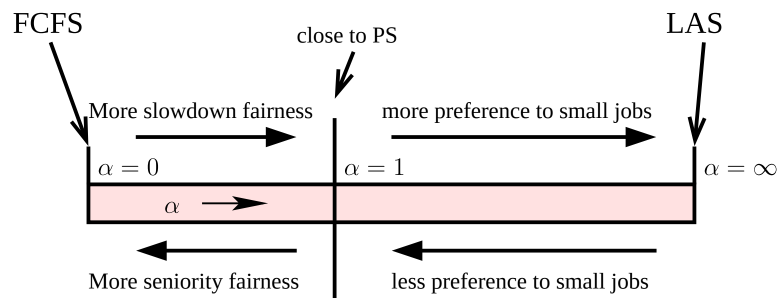

Figure 1: Variation of the PBS policy behavior.

- *Other well-known blind polices and mixtures*. Also, if we change the priority function slightly to $\mathrm{sgn}(\alpha)T/S^{\alpha}$, where $\mathrm{sgn}(\alpha)$ is the sign of α, then the domain of α is extended to the real set. In other words, for $\alpha < 0$, the scheduler chooses the one with minimum T/S^{α} instead of the maximum. Under this extension, the policy becomes a preemptive last-come first-served (PLCFS) policy as α tends to 0^{-}, and as α tends to $-\infty$ it becomes 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. This configuration is beyond the scope of this paper.

Figure 1 depicts the variation of behavior of the PBS policy as α changes.

In our implementation, an alternative priority function is used, by taking logarithm on both sides of Eq. (1):

$$p_i(t) = \log P_i(t) = \log T_i(t) - \alpha \log S_i(t), \tag{2}$$

where the base-2 logarithm is used. We expect a reduction of computational time by replacing the division and the power operation with two logarithm operations, one multiplication and one subtraction. In fact, both T_i and S_i are stored as integers rather than floats, and we can use some approximation methods to compute the base-2 integer logarithm. A very rough approximation can be done by searching for the highest bit of an integer, which is a single instruction on many architectures. Also, Eq. (2) brings us many nice properties that can be used to extend the policy; we shall explore this in Section 5.

It can be seen that the PBS policy scales. Look at Eq. (2). If we stretch the temporal line by a factor of K, and the sojourn times and processing times of all tasks are also scaled by K times, then their priority values are increased by a same constant $(1-\alpha)\log K$. For any fixed α, the decision of the scheduler remains the same on a scaled temporal line.

3 Behavior and Performance

Except for the extreme cases (e.g. FCFS and LAS), the behavior of the PBS policy is not quite clear. In this section, we identify some properties of this policy, and illustrate its dynamics by simulation.

For analysis purpose, we assume the time slices are infinitesimally small, as if at any time t, each task can take any given percentage of the processing time, referred to as the *processor share* of the task at t. The total processor share of all tasks is constant at any time. The context switch cost has to be ignored under this assumption.

An active task is called *scheduled* at time t if it receives a positive processor share *immediately after* t. (Note that it differs to say that a task *can be scheduled*, in which case we just mean the task is active.) 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 the identical priority value, which is greater than those of non-scheduled active tasks. If a scheduled task becomes non-scheduled after time t, we say that this task gets a *service interruption*.

Before analyzing the detailed behaviors of the PBS policy, let us first sketch them and illustrate with simulation results.

3.1 Behaviors of the PBS policy

Now we briefly describe how the PBS policy schedules tasks. We shall justify these descriptions in the following section.

- For $\alpha < 1$, a new task gets a slow start, and gains more and more processor share as time goes on. Younger (later arriving) tasks at some time may grab processing time more than their fair proportion, and could even interrupt the service of older tasks, but they have to eventually surrender all their processor share greater than their fair proportion back to the older tasks.

- For $\alpha > 1$, a new task immediately occupies the whole processor and interrupts all other tasks. As the priority value of the new task goes down, the scheduler will at some time resume the youngest interrupted task, then later on resume the youngest of the rest, and so on so forth, towards a fair proportion. These resumed tasks will not be interrupted again by any tasks already in the system. The new task may be recursively interrupted by a newer one before or after it begins to surrender the processor back to the older tasks.

- The case $\alpha = 1$ is a compromise between $\alpha > 1$ and $\alpha < 1$. A new arrival immediately gets an amount of processing time probably more than its fair proportion, but not the entire processor, and then begin to surrender it; however it will not end in surrendering all its share, but only the amount that exceeds the fair proportion.

We give the simulation results to illustrate the transient behaviors of the PBS policy and further demonstrate its ability to provide a smooth transition from the FCFS policy to the LAS policy. Suppose four tasks start at 0s, 1s, 3s, and 5s, with continuous demands of computational time of 4.5s, 2.5s, 3s, and 2s, respectively. Figure 2 illustrates the processor share obtained by each task from time to time. The abscissa is the elapsed time (between 0 and 12 seconds) since the arrival of the first task, and the ordinate proportionally delimits processor share of the tasks. As we can see in Figure 2, at $\alpha = 0.01$ ($\alpha = 100$), the PBS policy is almost identical to the FCFS (LAS, respectively) policy. The difference is, however, for $\alpha = 0.01$, every task still receives a tiny amount of processor share immediately after its arrival, which is too small to show up in the figure.

Figure 3 shows arrivals of four new tasks arrive 2.5 seconds after the arrival of the first task. Note that in all cases, the oldest task #1 gets service interruption for a while, but it is eventually resurrected. All tasks finally get equal proportions.

3.2 Properties of the PBS policy

We now itemize the detailed properties that pertain to the PBS policy. These properties may provide a rough idea on what kind of tasks are scheduled with how much processor share

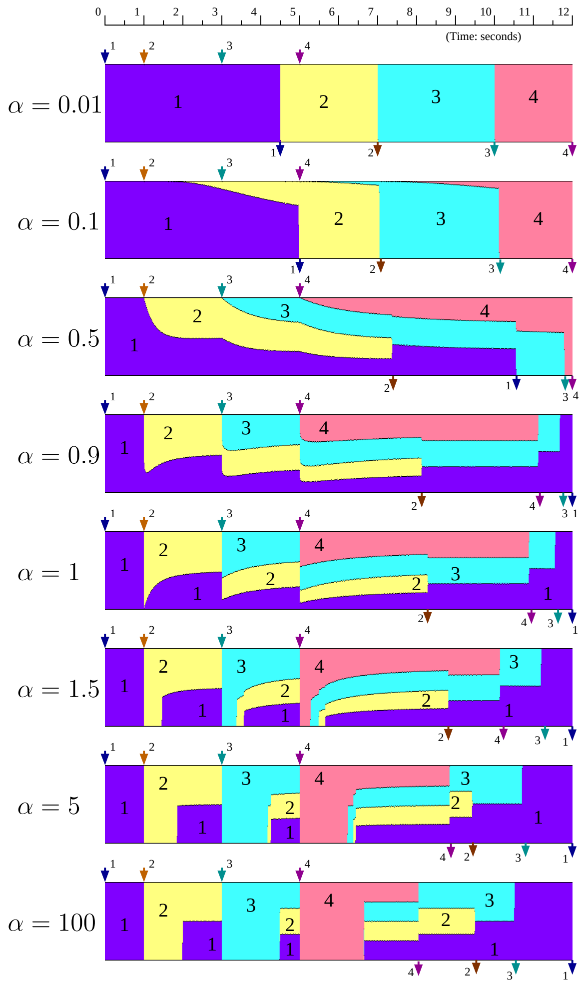

Figure 2: Illustration of processor share that each of four tasks obtains in a 12 second interval.

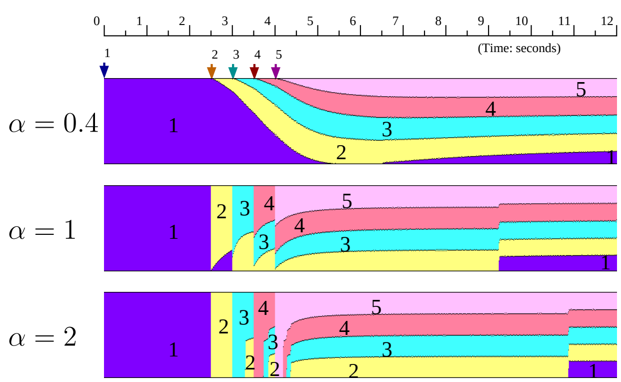

Figure 3: A persistent task sees four new arrivals at 2.5s, 3s, 3.5s, and 4s, respectively, and gets service interruption.

granted, at any clock time t.

1. *Finite priority.* An active task that has ever received a positive amount of processing time gets a finite priority value.
2. *Faster priority increase for non-scheduled.* A non-scheduled active task has a linearly increasing priority value for $\alpha < \infty$. The priority value of a scheduled task either decreases, or increases sub-linearly for $0 < \alpha < \infty$.
3. *Basic fairness toward seniors.* At any time, an older task (which arrives earlier) receives total processing time no less than a younger one does.
4. *Immediate service.* For $\alpha > 0$, a new task is immediately scheduled (i.e., it immediately gets processing time).
5. *Everybody contributes to the newcomer.* For $\alpha > 0$, all scheduled tasks should experience a decrease of processor share at the arrival of a new task, compared with the scenario without the new arrival.
6. *Quick start with a large α.* For $\alpha > 1$, a new task takes all processor share at the time of arrival (i.e., all other scheduled tasks experience service interruption). The new task will gradually surrender the processing time back to others.
7. *Slow start with a small α.* For $\alpha < 1$, a new arrival does not take entire processor at the time of arrival.
8. *Everybody benefits from the departed.* When a task departs, all other scheduled tasks experience an increase of processor share, compared with the scenario without the departure.
9. *Newcomers always have share.* If a task is scheduled, all the tasks arrived later than this task also get scheduled. In other words, only the earlier tasks may experience service interruption.
10. *No deprivation with a large α.* For $\alpha \geq 1$, all scheduled tasks remain scheduled until a new task arrives.
11. *Possible deprivation from seniors with a small α.* For $0 < \alpha < 1$, the oldest scheduled task may become non-scheduled even if there is no new arrival.

We now briefly justify these properties. Properties 1 and 2 are obvious by the definition of the priority value. To show Property 3, we assume at some point of time, an older task receives same amount of processor share as a younger one. At this point, the older one should have a higher priority than the younger and then receive more processor share, since the older has a greater sojourn time. Property 4 is straightforward by contradiction: if a task arrives at clock time t and does not receive any processing time in the period $[t, t+\varepsilon]$, its priority value is infinite during this period, and therefore by definition it must have been receiving processing time before clock time $t+\varepsilon$ for whatever positive ε. To show Property 5, we first note that at least one task has to contribute some of its processing time to the new task during $[t, t+\epsilon]$, by Property 4. At $t+\epsilon$, the priority value of the contributing task then goes greater than that in the scenario without arrival. In order to keep being scheduled, all other scheduled tasks must also get the same higher priority value at $t+\epsilon$, meaning that each of them has to contribute some of their processing time during $[t, t+\epsilon]$. Properties 4 and 5 do not apply to the case that $\alpha = 0$ (i.e., the FCFS policy). As to property 6, since $T_i/S_i \geq 1$, as S_i tends to 0, we have $T_i/S_i^\alpha \geq 1/S_i^{\alpha-1} \to \infty$ for $\alpha > 1$, which means the priority value of a new task can be arbitrarily large, if the total processing time it has obtained since its arrival is arbitrarily small. In other words, new tasks get very high priority, and remain as the highest one for a while after its arrival (at least until $1/S_i^{\alpha-1}$ is less than the maximum priority of the other active tasks), so it will interrupt the service of all existing scheduled tasks. For property 7, we note that, the new task i must have a priority value that is no less than those of existing tasks in order to receive processing time, i.e., P_i is bounded below. Then, for $\alpha < 1$, $S_i/T_i \leq S_i^{1-\alpha}/P_i \to 0$ as $S_i \to 0$, which means that the processor share of a task is zero at its arrival and is very small just after its arrival, therefore it could not interrupt the existing services of all other tasks immediately. Property 8 can be shown with an argument similar to that of Property 5. We demonstrate Property 9 next and Property 10 in Appendix. Property 11 is justified in Figure 3 with $\alpha = 0.4$. (The task #1 becomes non-scheduled at 5s.)

Justification of Property 9. Let us consider two scheduled tasks in the system. At time t, let $T_i = T_i(t)$ and $S_i = S_i(t)$, $i = 1, 2$, be the sojourn time and attained service time of task i at time t, respectively. Note that $S_i'(t)$, the derivative of $S_i(t)$, is the processor share of task i at time t, which must be between zero and one ($S_i' = 1$ means that task i takes all the processor time). Clearly the derivative of $T_i(t)$ with respect to t is one. Suppose both tasks remain scheduled in the active list during $[t, t+\varepsilon]$ for $\varepsilon > 0$. Since we have $T_2 S_1^\alpha = T_1 S_2^\alpha$ in $[t, t+\varepsilon]$, taking derivative on both sides, we get

$$\alpha S_1^{\alpha-1} S_1' T_2 + S_1^\alpha = \alpha S_2^{\alpha-1} S_2' T_1 + S_2^\alpha$$

or

$$S_1^\alpha \left(\frac{\alpha S_1' T_2}{S_1} + 1 \right) = S_2^\alpha \left(\frac{\alpha S_2' T_1}{S_2} + 1 \right).$$

Replacing S_1^α with T_1 and S_2^α with T_2 since $T_1/S_1^\alpha = T_2/S_2^\alpha$, we obtain

$$\frac{S_1'}{S_1} - \frac{S_2'}{S_2} = \frac{1}{\alpha} \left(\frac{1}{T_1} - \frac{1}{T_2} \right). \tag{3}$$

Without loss of generality, we assume task #1 arrives earlier, i.e., $T_1 > T_2$. Therefore the right-hand side of Eq. (3) is negative for $\alpha > 0$ and we get

$$\frac{S_1'}{S_1} < \frac{S_2'}{S_2}. \tag{4}$$

Therefore, as long as $S_1' > 0$ in $[t, t+\epsilon]$, we have $S_2' > 0$ in $[t, t+\epsilon]$. In other words, Eq. (4) states that, if both tasks are scheduled at time t, the younger task must keep scheduled after t as long as the older task keeps scheduled. Therefore, if one of the two tasks becomes non-scheduled, it must be the older task (note again that the younger task is always scheduled upon its arrival). If neither task is scheduled at t, the priority value T_i/S_i^α is linearly increasing with a slope of $1/S_i^\alpha$. Because the older task has a greater attained service time S (property 3), its priority value would be increasing slower than the younger, therefore the younger task must be resumed earlier. Then we have Property 9.

Persistent tasks without arrivals and departures. Consider several persistent tasks and assume after time t, there is no new arrivals and departures. Due to Property 2, the priority value of non-scheduled tasks must increase faster than that of scheduled ones, therefore eventually all non-scheduled tasks would become scheduled. Then we can look at Eq. (3) that assumes two tasks are scheduled. The right-hand side of (3) converges to zero as T_1 and T_2 tends to ∞, and we have $S'_1/S_1 \approx S'_2/S_2$, or in other words,

$$\frac{S'_1}{S'_2} \approx \left(\frac{T_1}{T_2}\right)^{1/\alpha} \to 1 \quad \text{as } T_1, T_2 \to \infty,$$

because $T_1 - T_2$ is a constant. This result means that eventually the PBS policy will converge to a fair proportion configuration (i.e., processor sharing), as long as $\alpha > 0$, no matter how small or large the α is. For α tends to ∞, it is well known that LAS will eventually become PS if there is neither arrival nor departure (hence it is also called Foreground-Background Processor-Sharing, FBPS).

3.3 No starvation

One of the obstacles when using a priority scheduling is the possibility of starvation. Here, we consider the starvation problem in a system with finite users. After having fully received service, each user has non-zero thinking time before submitting his/her next request of service. Such a system is always stable in the sense that the accumulated work would never go to infinity. However, both FCFS and LAS policies have starvation problems. For FCFS, if one user submits a task that fails to end, all subsequent tasks submitted by others are starved. For LAS (as well as in the multi-level feedback scheduling implementation), if two users frequently submit small tasks, a long task submitted by a third user will never get to finish. For the PS policy, both situations would not result in starvation. The PBS policy is starvation-free, too, for $0 < \alpha < \infty$. First, other tasks will not be starved by a persistent task, as we have shown previously that all tasks will eventually reach a fair proportion configuration for $\alpha > 0$; for the latter situation, we need to assume that all users have a lower bound of average thinking time. If a long task is stalled by many aggressive users who periodically submit small tasks, its priority value will increase linearly toward infinity; no matter how small the tasks submitted by those aggressive users are, they would either eventually let the long task run before they themselves finish, or they are too small in total to grab all the processing time. Therefore, for either situation, the PBS policy would not starve tasks as long as $0 < \alpha < \infty$.

Although permanent starvation is avoided, a long term starvation is still possible with very small or large α. Therefore, for 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.

3.4 Response time

One of the main goals of building a system is its performance. Performance can be measured in many ways, for examples, the average response time and total throughput. In the setting of a finite number of persistent users with independent thinking time between tasks, a lower average response time usually implies an a higher throughput.

Other than the hardware, the scheduler plays an important role for the response time. A simple example is as follows. Suppose some tasks of the same size are waiting in a row to be scheduled. With the PS policy, all tasks finish at the same time and each of the tasks experiences a response time equal to the entire workload. With the FCFS policy, some tasks may end earlier and on average they experience a response time equal to only the half of the entire workload. Generally, the FCFS policy gives a less mean response time than the PS policy if the variability of the task sizes is small. On the other hand, if the variability is high, assuming random arrivals, the LAS policy outperforms the PS policy in terms of mean response time, especially when the workload is high and task sizes exhibit a heavy tail property [19]. If the task sizes follow an exponential distribution with unit variability, all blind policies result in an identical mean response time (this result can be derived from a conservation law [15]).

Without the assumption of random arrivals, these results may not accurately apply, but the general argument is still valid. With the PBS policy, due to its ability to emulate all these blind policies, we can reduce the mean response time in both high and low variability environments by setting proper values of α, but not for the case that the task sizes are close to exponentially distributed. Even so, there are other performance measures to be considered, for examples, the variation of response time and the responsiveness for interactive tasks. A greater α can help improve the responsiveness for interactive tasks whereas an α close to one would reduce the variation of response times.

3.5 Fairness

Asides 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. Some 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 [20]: the time spent in the system; and
- slowdown fairness [26]: 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 fair. Our 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 one, the policy is actually doing a trade-off between seniority fairness and slowdown fairness (as shown in Figure 1).

For all $\alpha \geq 0$, the PBS policy maintains the very basic fairness that each of PS, FCFS, and LAS policies does: at any time, an older task is always running ahead of a younger one, as stated by Property 3. Not all scheduling policies comply with this basic fairness; the PLCFS policy is the one that violates it most; both the Shortest Job First (SJF) and the Shortest Remaining Processing Time (SRPT) policies [22] are certainly not satisfying this one.

4 Implementation and Experiment Results

We implement the PBS policy in the Linux kernel version 2.6.15.7 with the single-processor configuration. The length of a time slice is set to the default value of 4 ms. The configurable parameter α is provided by a user program through a special system call. Figure 4 shows some of the experiment results for multiple kernel threads. It draws the dynamics of the processor share obtained by each thread over time. The processor share of each thread is averaged over a 50 ms window. Note that the processing time is computed by a user-space multi-threaded

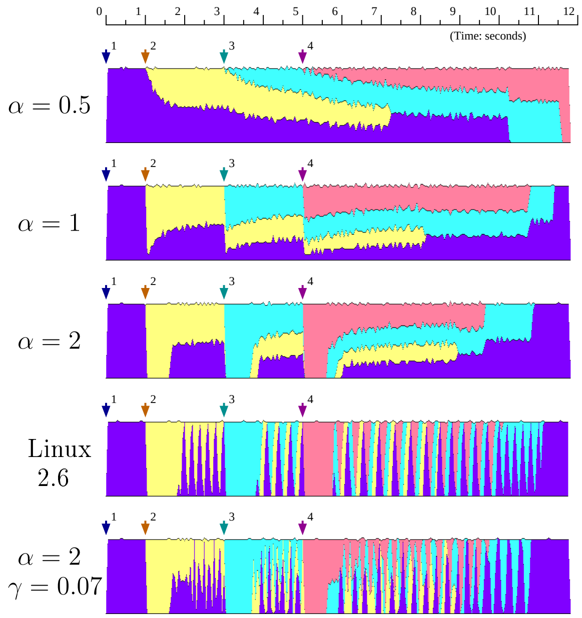

Figure 4: The processor share measured by a user-space test program, running under the modified Linux kernel with the PBS scheduler.

program itself who calculates how much work has been done, not through any stastistics from the kernel.

In Figure 4, four kernel threads start at 0s, 1s, 3s, and 5s, run roughly for 4.5s, 2.5s, 3s, and 2s, respectively, and then terminate, exactly like the settings in Figure 2. Comparing Figure 4 with Figure 2, we can see that the modified scheduler in the Linux kernel implements the PBS policy on a time-sliced system, with behaviors exactly as we have expected.

Upon implementation, some practical issues need to be considered. Here we discuss one of them and address the rest in the next section. We have defined that the sojourn time $T_i(t)$ of task i starts at its arrival time. But what is the arrival time $\tau_i(t)$ in a real system? Specifically, is it the time of its creation, or the latest time ready for scheduling? We can even set $\tau_i(t)$ to be the clock time of the latest activation/deactivation event of the entire system, and make the PBS policy equivalent to the PS policy. Our implementation, however, provides a more flexible approach, which is based on our process state model described next.

4.1 Process state diagram revisited

In modern multi-tasking computer systems, multiple processes (and kernel threads, also known as light-weighted processes, LWP) share the processing time preemptively with equally-spaced time slices, separated by timer interrupts. The scheduling decisions are made usually at the time when the processor is handling the timer interrupt, and at the time when process states change.

Traditionally, during the life time of a process, it switches between two states: the *active state* and the *inactive state*. An inactive (blocked) process is unable to run at present, usually because some system resources are not immediately available. An active process is the process currently running on a processor, or waiting in a queue to be scheduled. The process state diagram is shown in Figure 5. For particular operating systems, there may be some other system-specific states. For example, in some systems the entire memory space of a process

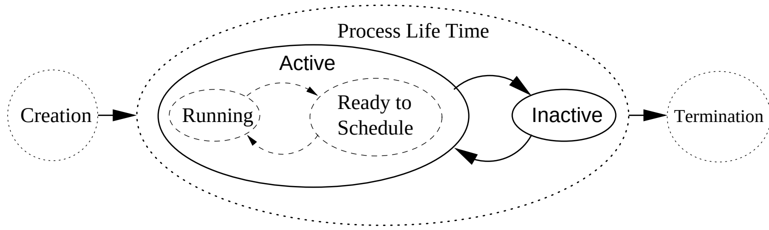

Figure 5: The traditional process state diagram.

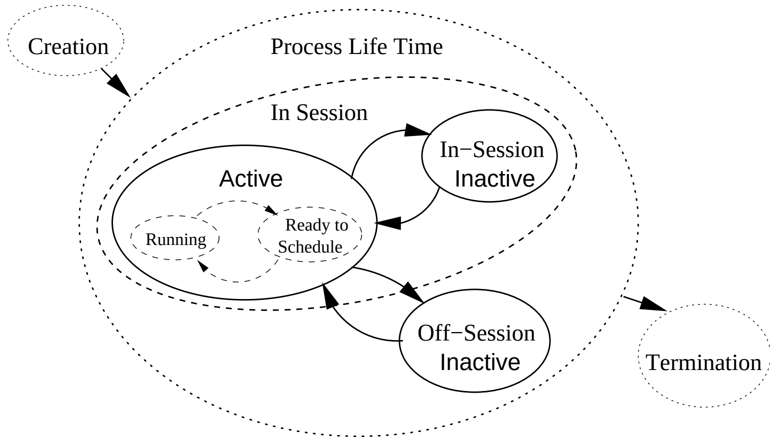

Figure 6: The process state diagram with sessions.

can be swapped out to disk by a mid-term scheduler, and therefore, one or more "swap-out" states are added into the diagram. Here we do not consider mid-term scheduling policies.

The diagram in Figure 5 is not sufficient to describe a phenomenon that becomes prevalent in recent years. Traditionally, a new process is created when a request arrives at a service system. This process terminates after all demands of this request have been fulfilled. However, the creation and termination of a process introduce a lot of overhead to the system as well as additional service delay to the request. Although the cost of creating a process has been minimized by many operating systems, there are still other kinds of expenses, for examples, the time to create a TCP connection and the time to read configuration files. To reduce such overhead and delay, many modern applications maintain persistent processes that do not terminate after each service, but instead remain inactive in the system for a relatively long time waiting for the next request. Hence, the inactive state is divided into two states: the *in-session inactive state* and the *off-session inactive state*. The process is in the in-session inactive state when it is awaiting other system resources to continue the processing of a request, or is in the off-session inactive state after it clears all services of the previous request. We refer to the duration of serving a single request as a *session*. The modified process state diagram is illustrated in Figure 6.

Unfortunately, most current operating systems do not provide any mechanism to accept notifications from applications stating that they are starting new sessions, much less to say forcing every application to send notifications. Therefore, the distinction between in-session and off-session inactive states, or in other words, the boundary of two consecutive sessions, is largely hidden from operating systems, and hence unavailable to the scheduler.

Nonetheless, using some heuristics, we can somehow guess boundaries of sessions. When a process is activated, if the last inactivation time, is much longer than the average inactivation time, we consider that the process is in an off-session inactive state, and a new session starts as soon as the process proceeds. Specifically, a new session starts at time t_{n+1}^A if the following condition holds:

$$t_{n+1}^A \geq t_{n+1}^I + \frac{\beta}{n} \sum_{i=1}^{n} (t_i^A - t_i^I),$$

where $[t_i^I, t_i^A]$ is the i-th inactive period. Quantity β is an adjustable threshold, which is usually greater than one. If β tends to 0, every active period is a session, whereas, if β tends to ∞, there is only one session in the entire life time of each process. One can in fact adapt the parameter β with real applications. Although not accurate, this heuristic probably suffices to be used by the scheduler. More complicated heuristics are possible; for example, we can take the variance of inactive periods into consideration for finding a better threshold, but the estimation cost escalates.

Not only do the server processes have multiple sessions as described earlier, but also do the desktop applications: a desktop user may be switching between different application windows, or just stop to think, when a session ends. By introducing the terminology of session, we are in fact clustering the active periods. By changing threshold β, we can get different session lengths. If a lower threshold β is used, the number of estimated session is then greater and each session becomes shorter.

With a β greater than zero, it is possible that there exist inactive periods within a session. Then we have two choices to compute the sojourn time. The first one is to include all inactive periods into the sojourn time, i.e., $T_i = t - \tau_i$; the other is to exclude them, i.e., $T_i = t - \tau_i - w_i$ where w_i is the past total inactive time in the current session. Just for notational convenience, we prepend a plus sign (+) to the β value to indicate that inactive periods are included, or a minus sign ($-$) to indicate otherwise. For example, $\beta = +\infty$ means the sojourn time is the time since its creation, whereas $\beta = -3$ means $\beta = 3$ with all inactive periods within the current session excluded from the sojourn time. For $\beta = 0$, clearly there is no difference between $+0$ and -0: both of them indicate that the sojourn time is the time since its last activation.

4.2 Experiment results

In this section we present various experimental measurement on real systems. Due to the limit of hardware availability and exclusive nature of such experiments, we unfortunately have only some aged platforms to use. The results scale, though, as we have explained. To reduce the interference from other running services in a system and non-conformability across systems, we use a self-compiled Linux From Scratch version 5.1.1 Linux distribution [7] with our modified kernel in all experiments.

We now present the results of three groups of experiments, under three different environments.

4.2.1 Group A: A simple computational model

In this group of experiments, we consider eight users doing computations simultaneously. Each computation task lasts approximately three seconds, and is divided into six segments. Between segments, the task program has a short inactive period (e.g., in order to wait for the data from the disk), which lasts about 10 ms. Between computation tasks, each user has a thinking time of 25 s on average.

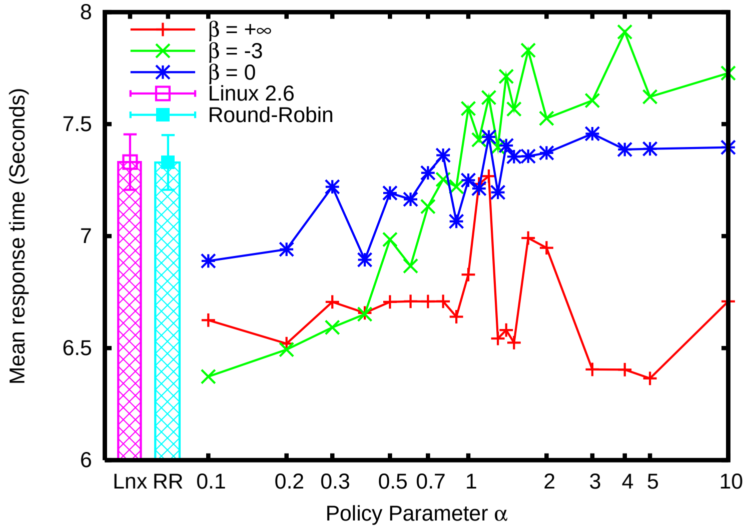

Figure 7: The mean response time for different α's and β's for the computational model. Note that the mean response time includes the task processing time of 3 s.

A multi-threaded user-space program emulates these activities and records the response time and throughput. It assumes that thinking periods are independent from each other, and exponentially distributed with a mean of 25 s. For each pair of α and β, we have a separate experiment session, measuring response time for a duration of 1200 s, with a 400s ramp-up time and a 100s ramp-down time. This group of experiments are performed on a Pentium III 550MHz desktop computer.

Figure 7 shows the mean response time for different α's and β's. For comparison, we also show the experimental results for the Linux 2.6.15.7 native scheduler, as well as a standard RR (round-robin) scheduler. For native and RR experiments, we repeat the same experiment for six times and compute the estimation error of mean response time across experiment repetitions. For the PBS policy, we do only one experiment for each pair (α, β); the estimation error is observed as the fluctuation of the curves (since the behavior of the PBS policy should be smooth over the variation of α).

As we can see from the results that the mean response time with the Linux 2.6 scheduler is roughly the same as that with the RR scheduler. The PBS scheduler, with $\beta = +\infty$, outperforms Linux and RR schedulers by 10-30% on mean additional delay (which is the mean response time minus the task processing time of three seconds). With $\beta = -3$ and small $\alpha < 0.5$, the PBS scheduler outperforms them by 30-40% on mean additional delay. For $\beta = 0$ and $\alpha < 1$, the PBS scheduler outperforms them by 0-10%.

What Figure 7 has not shown is the standard deviation of the response time. The standard deviation of response time is about 3.7 s for Linux scheduler, 3.6 s for RR scheduler, 6–7 s for $\beta = +\infty$, 4–5 s for $\beta = -3$ and 3.3–3.7 s for $\beta = 0$. Note that this deviation should not be confused with estimation errors of mean response time across experiments, which is quite small (less than 0.2 s).

Of the three choices of β, the mean response time is the most sensitive to α with $\beta = -3$. As α increases, the trend of increasing mean response time is significant. For $\beta = 0$, this trend is also observable. The better choices of the parameters are probably with $\beta = -3$ and $\alpha < 0.4$, in which case the mean response time is about 6.5 s, and the standard deviation of response

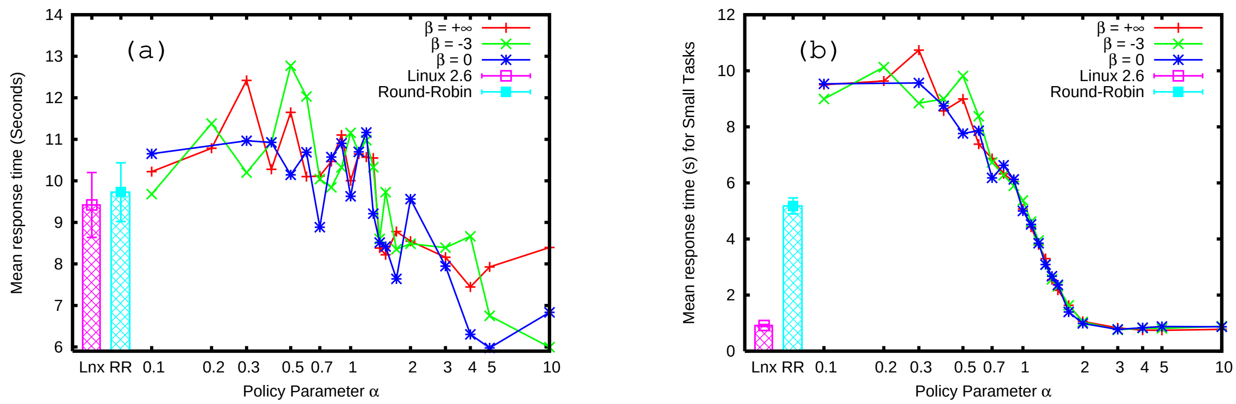

Figure 8: The performance of different α's and β's for the web server model. (a) The mean response time of all requests. (b) The mean response time of requests with small file sizes.

time is about 4–5 s.

4.2.2 Group B: A web service model

We perform this group of experiments with an Apache httpd server [3] (2.0.55) on an IBM Thinkpad T30. A total number of 30 clients generate HTTP requests with 10 s thinking time on average. The requested file sizes follow a Pareto distribution [6] with a shape index of 1.2. In order to make the web server a CPU-bound program instead of a network-bound one, dynamic web pages are generated by using a CGI program that reads random file contents from the disk with some additional processing, and returns a summary page to the client instead of the entire file. The additional processing time is proportional to the file size requested by the clients. The data is collected in the same way as in the group A experiments.

Figure 8 shows the mean response time for different α's and β's, compared with Linux and RR schedulers. Figure 8(a) shows the mean response time for all clients requests, and Figure 8(b) shows the mean response time only for small requests whose file sizes are less than the average requested file size. From these figures, it can be seen that a large α reduces the mean response time, in particular for small tasks. Although the Linux 2.6 scheduler is slightly better than RR in terms of the overall mean response time, it greatly boosts the responsiveness for small tasks. With large α's, the PBS policy outperforms the Linux and RR schedulers in terms of mean response time. For small tasks, with $\alpha > 2$, the PBS scheduler can provide as good responsiveness as the Linux scheduler does.

4.2.3 Group C: The TPC-W benchmark

In this group of experiments, we use the Java implementation of the TPC-W benchmark [24] by the PHARM research group at the University of Wisconsin-Madison [9]. We run all the programs (MySQL database server [17] 5.0.16, Apache Tomcat servlet container [4] 5.5.16, and the servlet and benchmark programs [9]) on the same hardware platform as in Group A. Figure 9 shows the mean response time for different α's and β's. As in Group A, we repeat the experiments six times for the Linux 2.6 and RR schedulers. The measurement time is 1800 s with 600s ramp-up and 300s ramp-down. The total number of clients is 30, each has an average thinking time of seven seconds.

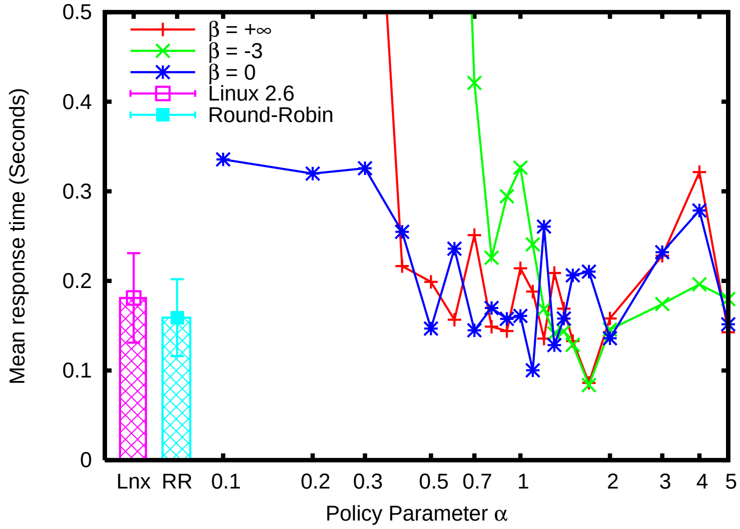

Figure 9: The mean response time for different α's and β's for the Java TPC-W implementation.

As we can see in Figure 9, the results of this group of experiments have a relatively larger estimation error than those of Group A. The RR scheduler performs a little better than the Linux scheduler. With large α's, the performance of the PBS scheduler is roughly in the same level as Linux and RR schedulers, and also appears to have a large estimation error. The performance becomes worse for small α's. Because of large measurement errors, it is hard to find a way to tune α for further improving the mean response time. The large measurement errors might be a result of putting so many different types of programs into a single box.

4.2.4 Comments on experiments

By now, we have shown in the real systems that the behavior and performance of the configurable PBS policy could be significantly changed by tuning the parameters, as demonstrated in Figures 4–9. We emphasize that performance changes are not necessarily, and should not be solely, expressed by the variations of mean response time and/or throughput. Note that, none of three experimental environments are typical and representative; in fact, there is no such thing as a typical system configuration. Even a TPC-W benchmark can only represent a certain class of e-commerce system [24], far narrower than the set of all environments that an operating system should be providing; this situation, however, adds more rationalness for creating a more flexible operating system, and in particular, a configurable operating system scheduler.

5 Additional issues and extensions

In this section, we go over the implementation issues not covered in Section 4, and discuss their relevancy to our PBS policy, as well as some possible extensions of the PBS policy.

5.1 Context switch cost

Context switch cost is always a concern when deciding the length of time slices and the scheduling algorithm, i.e., the frequency of timer interrupt. In Linux, the current default timer interrupt frequency is as high as 250Hz. The typical running time between context switches, however, is much lower, at about 10Hz as seen in Figure 4, unless affected by activation/deactivation events or user-defined `nice` values.

The PBS policy can be easily and elegantly adapted to reduce the number of context switches by adding a bonus to the priority value of the current process to Eq. (2). Some nice properties of this approach are:

- Younger tasks are less affected by this bonus, because their priority value changes faster as the sojourn time increases (note that function $\log$ is increasing and concave).
- For persistent tasks, the context switch frequency decreases as time goes on. More specifically, the time between two consecutive context switches increases linearly with respective to the sojourn times (as well as processing time) of these tasks.

The last property provides us a way to calculate the bonus of the current process. Suppose some tasks have been running for a while, and a task has received processing time of S. If ΔT is the desired time between two context switches, the bonus is given by

$$\gamma \approx \alpha \log\left(1 + \frac{\Delta T}{S}\right). \tag{5}$$

We justify the last property and Eq. (5) in the Appendix. Given $\alpha = 2.0$, if we expect that long tasks continously run for a period roughly equal to $\Delta T/S = 1/40$ of their processing time, we have

$$\gamma \approx \alpha \log(1 + 0.1) \approx 0.07.$$

Note that this is a very small value. An experimental comparison of the scheduler with and without this bonus is given in Figure 4. It can be seen in the figure that, with $\alpha = 2$ and $\gamma = 0.07$, the behavior of the PBS policy is fairly close to that of the Linux 2.6 native scheduler (except for the decreasing context switch frequency).

This property is not present in current systems like Linux, while it is kind of ideal philosophy to reduce the context switch cost. Note we do not use it (although implemented) in the experiments because our observations show that the context switch cost is minimum under the default Linux settings.

5.2 Weighted fair queueing

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` system call may introduce starvation to low priority tasks. With the PBS policy, we can add a finite user-specified offset δ (either positive or negative) to the calculated priority value. No starvation will be introduced, as explained in Section 3, but we must be extremely careful not to allow users to use large δ values. If we expect to give one task κ times of the processor share than other regular tasks, we should give it a priority offset of

$$\delta = \alpha \log \kappa, \tag{6}$$

or equivalently the ratio of processor share $\kappa = \exp(\delta/\alpha)$. With a δ given to each task, the PBS policy is easily adapted into a weighted fair queueing [12] flavor.

Note that the offset value δ has an exponential effect. A large δ might be assigned to some real-time applications — an exponentially high priority is somewhere between absolute priority and regular `nice` values.

5.3 Combined priority function

We can extend the function of the PBS policy to the following form

$$p_i(t) = \log T_i^\beta - \alpha \log S_i^\beta + \epsilon \left[\log T_i^{\beta'} - \alpha' \log S_i^{\beta'}\right],$$

with some fixed α' and β'. (We denote by $S_i^\beta := S_i^\beta(t)$ and $T_i^\beta := T_i^\beta(t)$ the processing time and the sojourn time of a session with given session threshold β, respectively. See Section 4 for the session threshold β.) This function is particularly useful with $\alpha' = 1$ and $\beta' \to \infty$, because the quantity inside the square braces is a measure of interactivity. With this setting, we still enjoy the configurability of the PBS policy while constantly adding a little favorability to interactive tasks. The favorability level can be controlled by quantity ϵ.

5.4 A summary

So far we have so many parameters, namely the policy parameter α, the session threshold β, the current process bonus γ, the user-specified priority offset δ, and the favorability level towards interactive tasks ϵ. Not all of them are, however, necessary to be optimized through experiments. The user-specified offset δ is given by the user, so we can just set upper and lower bounds according to Eq. (6). The current process bonus γ can be fixed to a proper value by Eq. (5). The session threshold β can be set to either 0, $\pm\infty$, or a value by looking at the behaviors of applications. We can determine the interactive task favorability level ϵ, if used, by looking at the behaviors of the original PBS policy under two pairs: (α, β) and (α', β'). The remaining parameter that might be tuned through experiments is the policy parameter α, which is the only parameter belonging to the policy. This paper does not intend to introduce a high-dimensional parameter set for users to optimize.

5.5 Scheduling complexity and the $O(1)$ scheduler

Scheduling complexity refers to the running time of a scheduler at each time it is invoked. If the scheduler goes through the information of all active processes, its complexity is at least $O(n)$ where n is the number of active processes. The FCFS and PS policies can each be implemented as $O(1)$ schedulers, e.g., it takes a constant time at each invocation. A priority scheduling policy with static priority values can be implemented as best as $O(\log n)$ through a priority queue algorithm (also known as heap sorting algorithm). With dynamically changing priority values, the scheduling complexity is $O(n)$ because the worst case is that the scheduler probably has no way to know the maximum priority value before going through every task. The priority value of the PBS policy is kind of dynamic (although it can be tracked analytically as in Appendix, it is a little complicated). For this reason, this policy can hardly be implemented with the priority queue algorithm. However, if an $O(n)$ algorithm is used, it can be easily implemented.

Algorithm 1 The $O(1)$ algorithm of the PBS policy.

Constants: $L > 1$, $M > 1$, and q (length of a time slice)
(Initialization at $t = 0$)

$\mathcal{A} = \{$`All active tasks`$\}$
$\mathcal{L} = \{L$ `tasks with greatest` $p_i(0)\}$

(At time t the scheduler is invoked)

$\mathcal{M} = \{M + L - |\mathcal{L}|$ `tasks from` $\{\mathcal{A} \backslash \mathcal{L}\}\}$
$\tilde{p}_i(t) = \log T_i\left(t + |\mathcal{A}| \cdot q/(2M)\right) - \alpha \log S_i(t)$ `for` $i \in \mathcal{L} \cup \mathcal{M}$
$\mathcal{L} = \{L$ `tasks with greatest` $\tilde{p}_i(t)\}$.
`Schedule` $\arg\max_{i \in \mathcal{L} \cup \mathcal{M}} \tilde{p}_i(t)$.

(At the arrival of task j)

$\mathcal{L} = \mathcal{L} + \{j\}$
$\mathcal{L} = \mathcal{L} - \{\arg\min_{i \in \mathcal{L}} p_i(t)\}$ `if` $|\mathcal{L}| \geq L$.

(At the departure of task j)

`If` $j \in \mathcal{L}$, $\mathcal{L} = \mathcal{L} \backslash \{j\}$.

Algorithmically, it is not possible to implement a priority scheduler with arbitrary priority values in $O(1)$ time. However, we have two ways to obtain an $O(1)$ scheduler with either approximation or *a priori* information:

1. The priority values are within a finite and small set, or they are forcefully discretized to a small set of values. Then, the scheduler can use an array of round-robin queues to implement an $O(1)$ scheduler with each queue corresponding to a particular priority value. The multi-level feedback scheduler is such an approximation. So is the scheduler in Linux 2.6.

2. The priority value in the future can be predicted or estimated (probably with errors), so we can distribute (or amortize) the scheduling cost to different invocations so that the (average) cost at each invocation is constant. We use this method to produce an $O(1)$ PBS scheduler. The algorithm is shown in Algorithm 1.

In Algorithm 1, two subsets of all active tasks are maintained: the tasks that have top L predicted priority values are in $\mathcal{L}$, and during each scheduling we choose another M tasks to form $\mathcal{M}$. Note that, if we pick M tasks each time from the set of all tasks $\mathcal{A}$, we would re-pick the same task every $|\mathcal{A}|/M$ repetitions. Therefore, we predict priority values at time $t + |\mathcal{A}| \cdot q/(2M)$, using the future sojourn time and the current processing time. Finally the scheduler schedules the task that has maximum current priority value within $\mathcal{L} \cup \mathcal{M}$. This algorithm is $O(1)$ since each step can be implemented in constant time, depends on constants L and M. We omit the details of the justification of this algorithm, but just to mention that an easy implementation is to set $L = 2$ and M to be a small number.

5.6 Related work

The PS (RR) policy is the oldest policy since the start of time-sharing era, which ends the dominance of the FCFS policy. It has been well studied [11] and ubiquitously applied. The Generalized Processor Sharing (GPS) extends the PS policy to support proportional sharing by weights tagged on tasks, instead of equal sharing [18]. Many implementations of GPS for operating systems and packet transmission networks have been proposed, for examples, Weighted Round-Robin (WRR) that uses unequal time slices, Weighted Fair Queueing (WFQ) [12], Worst-Case Fair Weighted Fair Queue (WF^{2}Q) [8] and Start-time Fair Queueing (SFQ) policies [14]. Readers may refer to [10] for a more complete list. Lottery Scheduling [25] is a stochastic implementation which gives tasks a number of tickets proportional to its weight, then randomly selects a ticket. The GPS policy is also extended and implemented in multi-processor environment, see [10] and references therein.

The LAS/FBPS/PS policy was first studied by Schrage [21]. A generalized feedback scheduling policy is proposed and studied by Kleinrock [15]. It recently received many attentions because its good performance with heavy-tailed task sizes [19]. The Preemptive Last-Come First-Served (PLCFS) policy is also received some attentions [2].

6 Conclusion

A generalized priority-based scheduling policy, PBS, is designed, implemented and evaluated in this paper. Not only can this policy emulate well-known blind policies like FCFS, PS, and LAS policies with a tunable parameter, but also does it emulate their mixtures of any levels in a continuous way. The performance of operating systems using this scheduling policy can be improved by empirically adjusting this parameter. This policy maintains fairness in three folds. First it provides a basic fairness that a task arriving earlier always go ahead of a task arriving later. Second, it eventually converges to equal share for each task if the tasks are long enough. Third, at the early processing stage of a task, the conflicts between performance and different classes of fairness can be balanced by the tunable parameter. This policy can also be easily modified to support weighted fair queueing, by adding priority offsets to tasks. The performance and tunability of an operating system using the PBS policy is demonstrated in this paper by simulations and experiments. In order to reduce its scheduling complexity, we also propose an $O(1)$ implementation of this policy.

References

[1] Linux Kernel Project. http://www.kernel.org/.

[2] ABATE, J., AND WHITT, W. Limits and approximations for the $M/G/1$ LIFO waiting-time distribution. *Operations Research Letters 20* (1997), 199–206.

[3] APACHE SOFTWARE FOUNDATION. Apache httpd. http://httpd.apache.org/.

[4] APACHE SOFTWARE FOUNDATION. Apache tomcat. http://tomcat.apache.org/.

[5] BANSAL, N. Achievable sojourn times by non-size based policies in a GI/GI/1 queue. http://www.research.ibm.com/people/n/nikhil/papers/blindnew.pdf, 2004.

[6] BARFORD, P., AND CROVELLA, M. Generating representative web workloads for network and server performance evaluation. In *SIGMETRICS/ PERFORMANCE '98:* (Madison, WI, November 1998), pp. 151–160.

[7] BEEKMANS, G. Linux from scratch. http://www.linuxfromscratch.org/lfs/.

[8] BENNETT, J. C., AND ZHANG, H. WF^2Q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE Infocom '96* (San Francisco, CA, March 1996).

[9] BEZENEK, T., CAIN, H., DICKSON, R., HEIL, T., MARTIN, M., MCCURDY, C., RAJWAR, R., WEGLARZ, E., ZILLES, C., AND LIPASTI, M. Characterizing a java implementation of tpc-w, in 3rd workshop on computer architecture evaluation using commercial workloads (caecw).

[10] CAPRITA, B., CHAN, W. C., NIEH, J., STEIN, C., AND ZHENG, H. Group ratio round-robin: $O(1)$ proportional share scheduling for uniprocessor and multiprocessor systems. In *Proceedings of the 2005 USENIX Annual Technical Conference* (Anaheim, CA, April 2005).

[11] COFFMAN, E., MUNTZ, R., AND TROTTER, H. Waiting time distribution for processor-sharing systems. *Journal of the Association for Computing Machinary 17*, 1 (1970), 123–130.

[12] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM '89* (Austin, TX, September 1989).

[13] FEDOROVA, A., SMALL, C., NUSSBAUM, D., AND SELTZER, M. Chip multithreading systems need a new operating system scheduler. In *Proceedings of 11th ACM SIGOPS European Workshop* (Leuven, Belgium, September 2004).

[14] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking 5*, 5 (1997), 690–704.

[15] KLEINROCK, L. *Queueing Systems Volume II: Computer Applications*. John Wiley & Sons, 1976.

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

[17] MYSQL AB. The mysql database. http://www.mysql.com.

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

[19] RAI, I. A., URVOY-KELLER, G., VERNON, M. K., AND BIERSACK, E. W. Performance analysis of LAS-based scheduling disciplines in a packet switched network. In *ACM SIGMETRICS* (Jun. 2004), pp. 106–117.

[20] RAZ, D., LEVY, H., AND AVI-ITZHAK, B. A resource-allocation queueing fairness measure. In *ACM SIGMETRICS* (Jun. 2004), pp. 130–141. New York, NY, USA.

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

[22] SCHRAGE, L. E., AND MILLER, L. W. The queue $M/G/1$ with the shortest remaining processing time discipline. *Operations Research 14*, 4 (1966), 670–684.

[23] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Applied Operating Systems Concepts*. John Wiley & Sons, 2000.

[24] TRANSACTION PROCESSING PERFORMANCE COUNCIL. Tpc-w: a transactional web e-commerce benchmark. http://www.tpc.org/tpcw.

[25] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of USENIX OSDI'94* (Monterey, CA, November 1994).

[26] WIERMAN, A., AND HARCHOL-BALTER, M. Classifying scheduling policies with respect to unfairness in an $M/GI/1$. In *ACM SIGMETRICS* (San Diego, CA USA, Jun. 13 2003), pp. 238–249.

A Deductions and Justifications

A.1 Analytic solution of processor share

Suppose there are no arrivals and departures since time t_0, and assume there are n tasks are in the scheduled set of active tasks. The sojourn time and processing time of task i at time t are denoted by $T_i = T_i(t)$ and $S_i = S_i(t)$, respectively. Suppose, at clock time t_0, $t_0 < t$, the attained service time $S_i(t_0)$ and the sojourn time $T_i(t_0)$ is known. We let

$$P(t) := \frac{S_1}{T_1^{1/\alpha}} = \frac{S_2}{T_2^{1/\alpha}} = \cdots = \frac{S_n}{T_n^{1/\alpha}},$$

and we have

$$P(t) = \frac{\sum_{i=1}^n S_i}{\sum_{i=1}^n T_i^{1/\alpha}}. \tag{7}$$

Then processing time of task k at time t, namely S_k, can be obtained:

$$S_k \;=\; T_k^{1/\alpha} P(t) = \left(\frac{T_k^{1/\alpha}}{\sum_{i=1}^n T_i^{1/\alpha}} \right) \sum_{i=1}^n S_i \tag{8}$$

Note that, since total processor share is one, $\sum_{i=1}^n S_i$ is a linear function of t with slope one. Taking derivative of S_k with respect to t, we get the processor share at time t:

$$\begin{aligned} S_k' \;&=\; \frac{1}{\alpha} \left(\frac{T_k^{1/\alpha - 1}}{\sum_{i=1}^n T_i^{1/\alpha}} - \frac{T_k^{1/\alpha} \sum_{i=1}^n T_i^{1/\alpha-1}}{\left(\sum_{i=1}^n T_i^{1/\alpha}\right)^2} \right) \sum_{i=1}^n S_i \\ &\quad + \left(\frac{T_k^{1/\alpha}}{\sum_{i=1}^n T_i^{1/\alpha}} \right) \qquad (9) \\ &=\; S_k \left[\frac{1}{\sum_{i=1}^n S_i} + \frac{1}{\alpha} \left(\frac{1}{T_k} - \frac{\sum_{i=1}^n T_i^{1/\alpha-1}}{\sum_{i=1}^n T_i^{1/\alpha}} \right) \right]. \qquad (10) \end{aligned}$$

Note again that T_i and $\sum_{i=1}^n S_i$ are both linearly increasing function of t.

Equations (8) and (10) provide us a way to efficiently estimate the progress of the PBS policy, which can drastically speed up the simulation program. Also from Eq. (8) we can see that S_k is a continuous, infinitely differentiable function of time t, unless S_k' hits the boundary of either zero or one (we have $0 \le S_k' \le 1$).

A.2 Property 10 of Section 3.3

Let us compute the derivative on Eq. (3) again. We finally obtain

$$\frac{S_1''}{S_1} - \frac{S_2''}{S_2} = \frac{1}{\alpha}\left(\frac{1}{T_1} - \frac{1}{T_2}\right)\left[\left(\frac{S_1'}{S_1} + \frac{S_2'}{S_2}\right) - \left(\frac{1}{T_1} + \frac{1}{T_2}\right)\right]. \tag{11}$$

As stated above, S_i is continuous and infinitely differentiable. Therefore, if task #1 is going to drop out from the scheduled list, it must happen at a point of time when $S_1' = 0$. At this point of time, from (4) we get

$$\frac{S_2'}{S_2} = \frac{S_1'}{S_1} - \frac{1}{\alpha}\left(\frac{1}{T_1} - \frac{1}{T_2}\right) = -\frac{1}{\alpha}\left(\frac{1}{T_1} - \frac{1}{T_2}\right)$$

and (11) becomes

$$\frac{S_1''}{S_1} - \frac{S_2''}{S_2} = \frac{1}{\alpha}\left(\frac{1}{T_1} - \frac{1}{T_2}\right)\left[\left(\frac{1}{\alpha} - 1\right)\frac{1}{T_2} - \left(\frac{1}{\alpha} + 1\right)\frac{1}{T_1}\right]. \tag{12}$$

For $T_1 > T_2$, the right-hand-side is positive if $\alpha \ge 1$. In other words,

$$\frac{S_1''}{S_1} > \frac{S_2''}{S_2} \quad \text{for } \alpha \ge 1.$$

Therefore, $S_1'' > 0$ as long as $S_2'' > 0$. Due to constraint of total processing share $\sum_{i=1}^n S_i' = 1$, we get $\sum_{i=1}^n S_i'' = 0$, and therefore $S_1 > 0$ if task #1 is the oldest task. In other words, for $\alpha \ge 1$, at point of time such that $S_1' = 0$, S_1' will be strictly increasing. In other words, if the oldest task gets a zero processor share at some point of time, its processor share will be increasing immediately, and this task would not drop out from the scheduled list. This proves Property 10.

A.3 Linearly increasing context switch intervals

Now we justify Eq. (5). Suppose n tasks have been running for T seconds, which is very long, therefore each of them gets almost a fair share, i.e., $S \approx T/n$. If we add γ to the log-priority value p (as shown in Eq.(2)) of the current process. After ΔT seconds, the log-priority value of the current task is

$$\gamma + \log(T + \Delta T) - \alpha \log(S + \Delta T)$$

and that of the non-scheduled others is

$$\log(T + \Delta T) - \alpha \log S.$$

When these two quantities agree, there would be a context switch. At that time, the elapsed time is

$$\Delta T \approx S\left(2^{\gamma/\alpha} - 1\right) \approx \frac{T}{n}\left(2^{\gamma/\alpha} - 1\right).$$

Then we get Eq. (5).

A.4 Priority offset for unequal share

Now we justify Eq. (6). Consider two tasks, #1 and #2. We add a priority offset to #1, with an amount of δ. Suppose after a long time the processor share for each task is stabilized. They have equal priority value if they both get scheduled, i.e.,

$$\delta + \log T_1 - \alpha \log S_1 = \log T_2 - \alpha \log S_2$$

Note that T_1 and T_2 are both very large, and their difference is constant. Then, we can assume that $\log T_1 \approx \log T_2$ (since the derivative of log approaches to zero). Then, we have

$$\delta = \alpha \log \frac{S_1}{S_2},$$

which is exactly Eq. (6) after processor shares are stabilized.