A User-Level Framework for Scheduling within Service Execution Environments

Travis Newhouse and Joseph Pasquale
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093–0114
{newhouse, pasquale}@cs.ucsd.edu

Abstract

To support application-specific quality of service for hosted services, a client of a compute utility requires the ability to schedule the processor resources supplied to its service. We present a user-level scheduling framework that operates in tandem with a standard kernel scheduler to support user-level policies for sharing processor resources. The scheduler operates by sampling the resource consumption of processes and limiting which processes are eligible for scheduling by the kernel. We present a Unix implementation of this framework and show that it can accurately control the rate of execution of compute-bound processes, with low computational overhead, despite its user-level operation. Finally, we demonstrate the scheduler's ability to enforce differentiated qualities of service for a Web-based message board service.

1. Introduction

Consider the scenario of a compute utility that offers machine time to run various types of applications and services (e.g., Web servers, network game servers), one per paying client, all on a single machine (or set of machines) [7]. The operating system of such a host divides the available processing power proportionally amongst a set of isolated execution environments for each client, perhaps based on the quality of service they are willing to pay. Although the host operating system schedules processor time across these environments, some clients need the ability to specify an application-specific scheduling policy for processor resources *within* an environment. Clients can benefit from user-defined scheduling policies in order to partition large shares of resources for delegation to other users [17], or to tailor the user-perceived quality of service provided to end users, by an application running within an isolation environment. The central question is, how can we provide this minimal type of quality-of-service control in a way that is simple to implement and easily deployable?

In prior research, though there have been many proposals for algorithms to allocate shares of processor bandwidth [13–15, 18, 19, 22], the algorithms generally rely on assumptions that typically cannot be satisfied in situations where users cannot modify the underlying operating system. Thus, our approach is to avoid changes to the kernel scheduler, but to enhance its operation via user-level mechanisms that realize alternative processor sharing policies. These mechanisms work at a higher level than the kernel scheduler, effectively guiding its scheduling decisions. A user-level design that relies upon only a few commonly supported features can be easily implemented and installed atop server operating systems, and without requiring administrator privileges. This simplifies experimentation with new algorithms to support application-specific scheduling policies.

A user-level approach for scheduling processor resources is not without its own challenges. The difficulties of developing user-level scheduling mechanisms lie in the lack of system information and the trade-off between accuracy in rate enforcement and operational overhead. A scheduler running at user level does not have access to the same information that is available to the kernel, such as notification when a running process blocks for I/O; likewise, a user-level mechanism cannot access timers with as fine a granularity as those available to the kernel, by which to preempt processes in an accurate and timely manner. Overhead is a potential problem because user-level scheduling must be performed by a process that itself must be frequently scheduled by the kernel to perform scheduling decisions.

In this paper, we present the design and implementation of a user-level scheduling framework that enables the development of user-level policies for sharing processor resources. The novelty of our design lies in that the user-level scheduler limits the decision space of the kernel scheduler, rather than making fine-grained scheduling decisions itself. We discuss a method for process selection that enforces proportional processor scheduling when incorporated into our framework. We experimentally evaluate our process selection method and show the trade-off between accuracy and overhead. Finally, we demonstrate the ability of our user-level scheduler to enforce a policy of processor consumption for a Web-based bulletin board service.

2. Design

The basis of our design is a two-level approach to scheduling in which the user-level scheduler makes coarse-grained decisions while the kernel scheduler makes fine-grained decisions. Our design relies on the kernel scheduler for fine-grained decisions because the kernel scheduler has intimate system knowledge that is unavailable to a user-level mechanism. The user-level scheduler's coarse-grained decisions restrict the kernel scheduler's decision space to a set of processes that are eligible to run according to a user-level scheduling policy.

Making scheduling decisions that select *multiple* eligible processes and relying on the kernel to make fine-grained decisions reduces overhead by taking the user-level scheduler out of the critical path of each fine-grained scheduling decision. It also prevents potential waste due to underutilization of the processor. Unlike the kernel scheduler, a user-level scheduler does not know, for example, when a process blocks for an I/O or sleep request. As a result, a user-level scheduler cannot select, in a timely fashion, another process to run in place of the blocking process. Thus, the processor may become idle until the user-level scheduler makes its next scheduling decision. On the other hand, in our two-level scheduling approach, the kernel scheduler will select a new process to execute from the remaining eligible processes, thereby ensuring the processor is utilized so long as there are processes that meet the eligibility requirements of the user-level scheduling policy.

Left unattended, however, the kernel scheduler may make scheduling decisions that do not promote the policy of the user-level scheduler. For instance, many kernel schedulers aim to provide each process with an equal share of the processor. This will clash with a user-level policy that seeks to give processes unequal shares. Thus, the user-level scheduler must take action to prevent the kernel scheduler from making decisions that hinder the fulfillment of user-level policies.

2.1. User-level scheduling framework

Our user-level scheduling framework divides time into equal-length *scheduling decision intervals*. At the beginning of each scheduling decision interval, the scheduler decides which processes are eligible to run during the interval. For the duration of the interval, the selected processes contend for processor time according to the kernel scheduler's actions. The number of processes that actually execute during the interval will then depend on the length of the interval and the maximum duration that the kernel allows a process to run at one time, as shown in Figure 1. The scheduling decision interval is effectively a higher-level "quantum" that is applied to a *group* of processes, and is the decision-making granularity of the user-level scheduler. Thus, the scheduling decision interval provides an upper bound on how far a process may exceed its allocated rate of execution. We allow a user-level scheduling policy to specify the length of the scheduling decision interval to meet its requirements for accuracy and overhead.

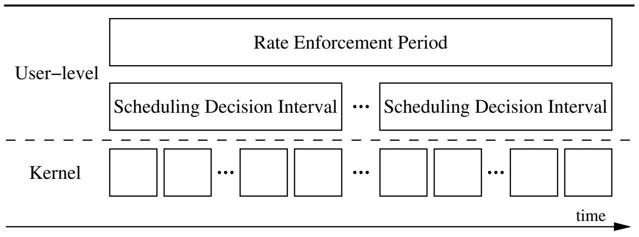


Figure 1. Relationship of time scales. The kernel scheduler makes fine-grained decisions to schedule single processes. The user-level framework's scheduling decision interval consists of a number of the kernel's execution slots. At the highest level of granularity, the process selection algorithm divides time into rate enforcement periods that are integer multiples of the scheduling decision interval

Also shown in Figure 1 is the grouping of a number of scheduling decision intervals into a *rate enforcement period*. This is actually not an inherent part of the framework, but rather, is defined as an abstraction for measuring fairness in the sample user-level scheduler that we have built, and which we further describe in Section 2.2.

The user-level scheduler influences the kernel scheduler by specifying to the kernel which group of processes are eligible to run. A process belongs to exactly one of either the *ready group* or the *suspended group* for the entire duration of each scheduling decision interval. At the start of each scheduling decision interval, the user-level scheduler calls on a procedure that implements a *process selection algorithm* to assign each process to one of the two groups. Thus, the process selection algorithm is what ultimately defines the policy of the user-level scheduler.

The user-level scheduler carries out the process selection algorithm's group assignments using the (assumed) suspend and resume mechanisms provided by the underlying operating system. Just as if there were no user-level scheduler on the system, the task of the kernel scheduler remains to select a ready process to execute on an available processor. The task of the user-level scheduler, by way of its process selection algorithm, is then simply to decide whether a process belongs in the ready group or the suspended group.

2.2. Process selection algorithm

We have developed a process selection algorithm that supports proportional share scheduling policies. The process selection algorithm divides time into equal-length *rate enforcement periods* that are integer multiples of the scheduling decision interval (Figure 1). The task of the process selection algorithm is to ensure that, at the end of each rate enforcement period, any process that is ready-to-run for the entire period will execute at exactly the rate to which it is entitled. We note that at any point *during* a period, a process's rate of execution may be over or under its allocated share. Thus, a shorter rate enforcement period yields more accurate control than a longer one.

The rate enforcement period provides an intuitive notion of the deadline by which fairness guarantees will be met. The length of the rate enforcement period can be tuned, within practical limits, to meet the fairness requirements of a particular policy. For example, a policy may specify that the dispatch latency for a process is not more than T time units. If each process executes at least once per rate enforcement period, setting the rate enforcement period equal to T will provide an average dispatch latency of T and a worst case latency of $2T - a$, where a is the amount of processor time a process deserves during one period.

The rate enforcement period also provides a means of "forgetting the past". If the process selection algorithm considers a process's rate of execution over the process's entire lifetime, then a process that blocks for a long time may execute in the future at a rate that greatly exceeds its allocated rate. By defining rate guarantees in terms of the rate enforcement period, the process selection algorithm need only ensure that a process's rate of execution is correct at the end of each rate enforcement period. By a process receiving its allocated share within each period, it follows that it receives the correct share over its lifetime. Moreover, a process that voluntarily waives its share of the processor during a period (e.g., by blocking on a sleep or I/O request) is not entitled to extra processor resources in a future rate enforcement period.

Under a proportional share scheduling policy, each process p_i has a share, s_i, which is a positive integer. The process selection algorithm ensures the process's rate of execution is proportional to its share as follows. A process's weight, w_i, is its share divided by the sum of the shares of all processes:

$$w_i = \frac{s_i}{\sum_j s_j}$$

A process's allocation, a_i, is the amount of processor time to which the process is entitled during a rate enforcement period of length R time units:

$$a_i = w_i \cdot R$$

The rate at which a process may consume the processor is one allocation per rate enforcement period.

Our process selection algorithm tries to maintain a process's utilization close to its share of the processor. A process's utilization is the ratio of its processor time to the wall time over which that processor time was received. For a process p_i, let $c_i(t)$ equal the processor time consumed by the process from the beginning of the current rate enforcement period until the beginning of interval t. The utilization, $u_i(t)$, at the beginning of interval t is the processor time consumed by the process since the beginning of the current rate enforcement period divided by the elapsed wall time since the beginning of the current rate enforcement period:

$$u_i(t) = \begin{cases} 0 & \text{if } t = 0 \\ \frac{c_i(t)}{t \cdot S} & \text{if } t > 0 \end{cases}$$

where S is the length of the scheduling decision interval and t indexes the intervals of a rate enforcement period from 0. At the beginning of a scheduling decision interval, the process selection algorithm assigns to the ready group each process whose utilization is less than its weight:

$$READY(t) = \{p_i : u_i(t) < w_i\}.$$

Otherwise, the process becomes a member of the suspended group.

3. Implementation

We implemented our user-level scheduler as a (user-level) daemon, `superd`, that runs on both FreeBSD and Linux operating systems. The implementation uses standard mechanisms found on most UNIX-like systems, and should also be portable to other operating systems that provide similar functionality. An additional command named `supervise` launches an executable as a process scheduled by the user-level scheduler. The `supervise` utility contacts `superd` to register a share value and the process identifier (PID) of itself before loading the specified executable into the registered process (e.g. using the `exec()` system call). Upon receipt of a new PID, `superd` sets the process's "nice" value to be lower than its own, to ensure that the process selection algorithm will run promptly during each scheduling decision interval. All processes under `superd`'s control execute with the same nice value.

The `superd` process sets a real-time interval timer to expire at the end of every scheduling decision interval. When the timer expires, the signal handler measures the processor consumption of each process and invokes the process selection algorithm to make scheduling decisions. The processor usage for a process is obtained via the process filesystem (`procfs`), a standard extension available on most UNIX-like systems. `procfs` represents each active process as a directory containing files that provide an interface to kernel data structures for that process. `superd` reads only from a status file that provides a process's user and system execution times.

After the processor consumption of each process has been measured, the process selection algorithm makes any necessary scheduling decisions. Our proportional-share process selection algorithm checks whether the current scheduling decision interval marks the end of a rate enforcement period. If a period has ended, the algorithm resets any per-period variables, such as the processor time consumed by each process during the current period. Then, it assigns each process to either the ready group or suspended group for the next interval.

To move a process from the suspended group to the ready group, `superd` sends a `SIGCONT` to the process. This notifies the kernel scheduler that the process is eligible for execution. To move of a process from the ready group to the suspended group, `superd` sends a `SIGSTOP` to the process. Sending a `SIGSTOP` signal to a process will suspend the process, taking it out of the kernel's ready queue, until a `SIGCONT` is sent to the process at a later time. Further, a process cannot block, catch, nor ignore a `SIGSTOP` signal, and therefore cannot avoid the control of `superd`.

4. Evaluation

Our evaluation consists of an experimental analysis of our scheduler, and a characterization of its accuracy and overhead. We performed all experiments on an unloaded machine to minimize the effects of external system load. The machine is a single 2.2 GHz Intel Pentium 4 processor with 512 MB of memory running the FreeBSD 4.8 operating system.

4.1. Process selection algorithm

To evaluate accuracy, we ran four compute-bound processes with shares $\{1, 2, 3, 4\}$ (corresponding to fractional processor rates of 10%, 20%, 30%, and 40%, respectively) and recorded the progress of each process at the end of each rate enforcement period. We set the rate enforcement period to 1 second and tested with scheduling decision intervals of 10, 20, 50, and 100 milliseconds. Each test ran for just over 1 minute, recording measurements for approximately 65 periods. Figure 2 shows each process's actual share of the processor and relative error at the end of each rate enforcement period when using a 50 millisecond scheduling decision interval. We discarded the first five periods of the test to remove (minor) start-up effects and focus on behavior in equilibrium. To summarize the results for each interval length tested, the root mean square of relative errors are 0.68%, 0.86%, 1.4%, 2.2% for scheduling decision intervals of 10, 20, 50, and 100 milliseconds, respectively.

4.2. Overhead

The overhead of our user-level scheduler warrants considerable attention. One might expect that the overhead of a user-level scheduling mechanism is unacceptably high for practical use. However, because the user-level scheduler simply makes high-level decisions to guide the kernel-level scheduler, rather than replacing it and making low-level decisions itself, the expected overhead is significantly reduced. We evaluate overhead by showing how our implementation scales with the number of scheduled processes. To test, we compare the total running time of a number of compute-bound processes with equal shares when scheduled by the kernel scheduler and the user-level scheduler. Figure 3 depicts how the overhead of our user-level scheduler scales with the number of processes that it schedules. The nearly linear scaling coincides with the linear complexity of the process selection algorithm that must calculate and compare the utilization of each process at each scheduling decision interval to decide which processes belong in the ready group. This test indicates that our user-level scheduler, coupled with a simple linear complexity process selection algorithm, can achieve acceptable overhead when using a 50 millisecond scheduling decision interval. Devising process selection algorithms with sub-linear complexity may result in user-level schedulers with lower overhead for smaller scheduling decision intervals.

4.3. Accuracy

We next evaluate the accuracy of our user-level scheduler by testing it with a set of non-equal shares. We chose a share distribution that illustrates how the user-level scheduler handles both a skewed distribution of shares and a set of shares that result in allocations that are impossible to accurately fulfill at certain scheduling decision intervals. The processes in the experiment all run the same compute-bound workload. The user-level scheduler uses a rate enforcement period of 1 second, and we repeat the experiment for scheduling decision intervals of 10, 20, and 50 milliseconds. We present the results as a plot of each process's absolute error during the first 30 periods of the experiment. The absolute error value is a mean calculated over 10 runs of the experiment. We limit the graphs to 30 periods for sake of resolution, and note that any stabilizing effects take place within that time frame.

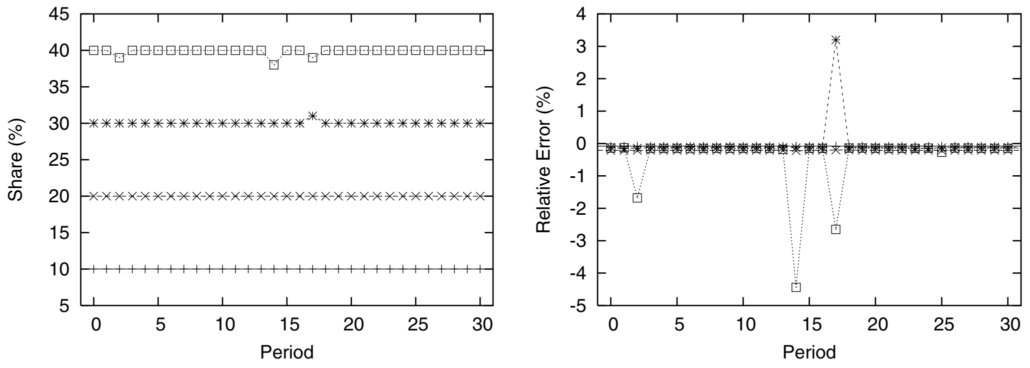


Figure 2. Process share and relative error for a 50 ms scheduling decision interval

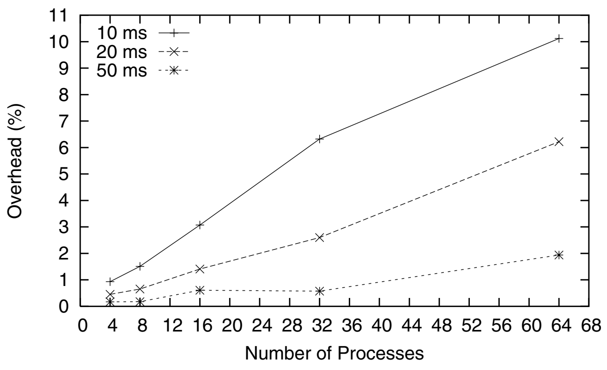


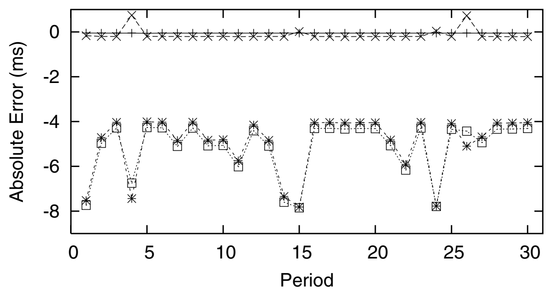
Figure 3. Runtime overhead using various scheduling decision intervals

Before discussing the results, we first point out an artifact of our implementation. By using a fixed length rate enforcement period over which the accuracy of the user-level scheduler is evaluated, the overhead induced by the user-level scheduler detracts from the total processing time available to the processes. For example, if the user-level scheduler causes 5 milliseconds of overhead during a rate enforcement period, that time is obviously not available to the processes. Additionally, if there is system load external to the user-level scheduler, such as a system process not under the control of the user-level scheduler, the execution time consumed by that load cannot be assigned to a process. This has a particular effect on the process with the highest share. Due to the kernel scheduler's bias toward lower-share processes, we have observed that the process with the highest share "absorbs" all of the user-level scheduler's overhead and external system load.

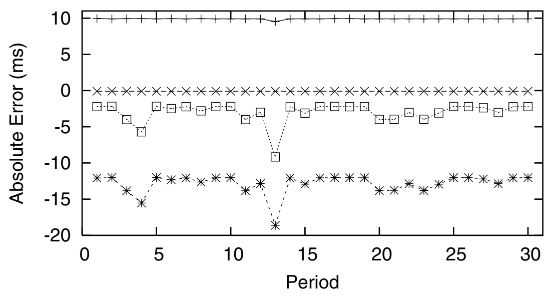
For this experiment, we chose the skewed distribution of shares $\{1, 4, 95\}$ which corresponds to allocations of 10, 40, and 950 milliseconds for a 1 second rate enforcement period. Figure 4 shows the results when using the three different scheduling decision intervals. With a 10 millisecond scheduling decision interval each process's allocation is an integer multiple of the interval length, and it is feasible to have near perfect accuracy. In fact, as shown in Figure 4(a), P_1 and P_4 receive exactly their allocation in most periods. P_{95} falls short of its allocation by roughly 5 milliseconds. This compares sensibly with the amount of overhead we observed when testing the user-level scheduler with 4 equal-share processes. We attribute the periodic spikes in P_{95}'s error to external system load.

When we increase the scheduling decision interval to 20 milliseconds, we see how the user-level scheduler behaves as conditions begin to stray from the ideal. In this case, P_1's allocation is less than the scheduling decision interval. As shown in Figure 4(b), the user-level scheduler does the best that it can. It must assign P_1 to the ready group for at least one interval. And, while it can assign P_4 to the ready group at the same time, in hopes that the kernel will split the processor time between the two processes, the user-level scheduler cannot guarantee that P_1 will not execute for the entire 20 millisecond interval. In fact, P_1 does execute for an entire interval, as exhibited by the 10 millisecond absolute error. Based on the kernel's bias for lower-share processes and P_4's perfect rate of execution, we can also assume that P_4 executes uncontested for exactly two scheduling decision intervals. The remaining 940 milliseconds minus any overhead is consumed by P_{95}. Notice that the sum of errors is about 2.5 milliseconds, which is in range with the overhead observed for 4 equal-share processes scheduled at this interval length. Also of note, P_{95}'s absolute error is exactly the amount of overconsumption by P_1 plus any overhead.

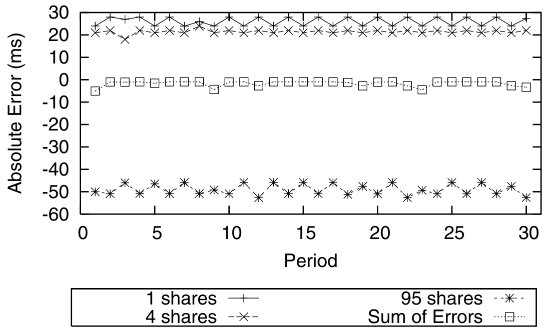
Finally, Figure 4(c) shows how the user-level scheduler handles two processes with allocations that are less than the 50 millisecond scheduling decision interval. In this test, the kernel schedules P_1 and P_4 during the first scheduling decision interval in a ratio that is inverse to their shares—P_1 receives nearly 40 milliseconds and P_4 receives slightly more



(a) 10 ms scheduling decision interval



(b) 20 ms scheduling decision interval



(c) 50 ms scheduling decision interval

Figure 4. Mean absolute error for processes with shares {1, 4, 95}, using a 1 second rate enforcement period

than 10 milliseconds. This is an attempt by the kernel scheduler to correct the user-level policy's desired share distribution. As a result, both processes' utilizations are greater than their weights at the beginning of the second interval. P_1 has exceeded its allocation by almost 30 milliseconds, and is therefore assigned by the user-level scheduler to the suspended group for the remainder of the period. P_4 remains in the suspended group until the first 250 milliseconds of the period have elapsed. At that point, its utilization drops below its weight, and the user-level scheduler allows it to run again. This time, the kernel schedules P_4 for the entire 50 millisecond interval, for a total execution time of approximately 60 milliseconds in the period, or 20 milliseconds more than its share. While P_{95}'s allocation is an integer multiple of the interval length, it executes for approximately 50 milliseconds less than its allocation, an amount equal to the overconsumption of the other two processes.

5. Example: Web-based bulletin board

The experiments in the preceding section characterized the accuracy and overhead of our user-level scheduler when applied to a synthetic, compute-bound workload. In this section, we demonstrate the utility of the scheduler when applied to a real service that performs computation and I/O. For our experiment, we use the RUBBoS benchmark that implements a Web-based bulletin board using PHP and a database [2]. The benchmark also provides a client workload driver that generates requests.

The experimental setup consists of a Web server, a database server, and three client workstations. The Web server is a 2.2 GHz Pentium4 processor with 512 MB of memory running FreeBSD 4.8 and Apache 2.0.48 configured with the "prefork" MPM and PHP 4.3.4. The database server and client machines are dual-Pentium III 600MHz processors with 1024 MB of memory running the Linux 2.4.20 kernel. The database server software is MySQL 3.23.58. A 100 Mbps and 1000 Mbps switched Ethernet connects the machines.

5.1. Experiment

We host three instances of the bulletin board Web site on the Web server machine by running a distinct instance of Apache on three different ports. Each instance of the Apache server is configured to use at most 50 processes, which Apache automatically regulates according to load. Our goal is to demonstrate the ability of the user-level scheduler to enforce different qualities of service for each instance of the bulletin board (e.g., to serve different classes of users). The scheduler uses a 100 millisecond scheduling decision interval to enforce a share distribution of $\{1, 2, 3\}$.

We first measure how the kernel schedules the Web servers by feeding requests from the client workstations. Each workstation uses 325 simultaneous clients to drive one of the three bulletin board Web sites. The number of clients was selected experimentally to achieve highest total throughput. The throughputs, measured in requests per second by the workstation machines, for the three Web sites are {29, 30, 40}. The kernel scheduler shares the processor roughly evenly with the three Web sites. Next, we employ our user-level scheduler to isolate the performance of the three different Web sites. Again, we generate a request workload from the client workstations using 325 simultaneous clients. The throughputs we measured are $\{18, 35, 53\}$

requests per second. Our user-level scheduler is capable of sharing the processor in the proportions that we specified and consumed only 3.2% of the total processor.

5.2. Remarks

As described in Section 4.2, the overhead of our user-level scheduler implementation scales linearly with the number of processes being scheduled because we must check the processor consumption of each process at the end of each interval. To achieve maximum throughput, we configure Apache to use a large number of processes. The same level of concurrency might also be achieved by using threads executing within one or a small number of processes. Development of a multi-process, multi-threaded Web server by the Apache group is in progress, but was not sufficiently operational at the time of this writing. Such a server configuration will significantly reduce the number of processes that our user-level scheduler must monitor.

Because we ran Apache as multi-process application, we had two choices as to how to impose our user-level scheduler on the Apache Web servers. We could modify Apache to notify the scheduler each time a new process was created, or we could monitor the processes created by Apache. We chose the latter so that we did not need to modify Apache. We configured the scheduler to update the processes associated with each Web server at the end of each 1-second period, and ran each Apache server under a distinct user account. To perform the update, the scheduler selects all the processes belonging to the user under which the Web server is running.

6. Related Work

Much work has been done in the area of process scheduling and resource control for performance isolation. Process scheduling algorithms for operating system kernels include proportional share algorithms [14, 19, 22] and algorithms that guarantee rates of execution [13, 15, 18]. Resource control mechanisms to support isolated execution environments have been developed for both operating system kernels [6, 8, 20, 21] and language-level execution systems [5, 11].

Scheduler activations [3] promote kernel and user-level cooperation to improve performance of user-level threads on a multi-processor. This kernel-supported mechanism enables the kernel to share processor availability information with a user-level thread scheduler to improve scheduling decisions and thread concurrency.

Arpaci-Dusseau and Arpaci-Dusseau introduce a gray-box Information and Control Layer that leverages what is known and can be inferred about the behavior of black-box kernel abstractions to gather information and influence system control without modifying the operating system kernel [4]. The authors also describe a technique for user-level scheduling that bases admission control on physical memory usage.

Chu and Nahrstedt augment the kernel scheduler to schedule soft real-time applications by using the real-time priority mechanism available in some forms of Unix to implement a reservation-based scheduler [10]. Adjusting the real-time priority of process can be performed from user space, but requires administrator privilege on most multi-user operating systems.

Chang et al. implement a user-level sandbox that provides a process with an execution environment possessing a subset of the resources provided by the physical machine [9]. To determine progress, they statistically estimate the time a process spends on the ready queue.

Others describe techniques for performance guarantees that interpose mechanisms in server communication subsystems [1, 23]. The techniques assume that network resources (e.g. network bandwidth, socket queues) are the bottleneck.

Other works have taken a control-theoretic approach to controlling application execution in which a feedback loop manages application resources. This requires modifying the application to report progress [12], or an understanding of the operation and performance goals of the application [16].

7. Conclusions

For shared computing platforms that provide services to clients, users can benefit from a mechanism for processor resource control to administer differentiated levels of service and performance isolation. We presented a user-level scheduling framework that enables the implementation of proportional share (and other) policies and requires no modifications to the underlying operating system kernel or the scheduled applications. We presented experiments that demonstrate that our scheduler meets the goals of enabling user-level enforcement of rate limits with low overhead and reasonable accuracy. Further, we have shown the feasibility of our user-level scheduling framework by using our scheduler to manage the processor resources of a Web-based bulletin board.

Our current work has focused on controlling compute-intensive processes. In the future, we plan to develop an improved algorithm for user-level proportional share scheduling that takes better account of I/O.

References

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. User-level QoS-adaptive resource management in server end-systems. *IEEE Trans. Comput.*, 52(5), 2003.

[2] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic Web site benchmarks. In *Proceedings of the IEEE 5th Annual Workshop on Workload Characterization*, 2002.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109. ACM Press, 1991.

[4] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 43–56. ACM Press, 2001.

[5] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*. USENIX, 2000.

[6] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. USENIX, 1999.

[7] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale services. In *Proceedings of the First Symposium on Network Systems Design and Implementations (NSDI)*, 2004.

[8] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting quality of service into a time-sharing operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 15–26. USENIX, 1999.

[9] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 25–36. USENIX, 2000.

[10] H.-H. Chu and K. Nahrstedt. A soft real time scheduling server in UNIX operating system. In *Proceedings of the 4th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 153–162. Springer-Verlag, 1997.

[11] G. Czajkowski and T. von Eicken. JRes: a resource accounting interface for Java. In *Proceedings of the conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 21–35. ACM Press, 1998.

[12] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 247–260. ACM Press, 1999.

[13] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 261–276. ACM Press, 1999.

[14] P. Goyal, X. Guo, and H. M. Vin. A hierarchial CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[15] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211. ACM Press, 1997.

[16] Y. Lu, T. F. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for QoS guarantees and its application to differentiated caching services. In *Proceedings of the 10th International Workshop on Quality of Service*, 2002.

[17] T. Newhouse and J. Pasquale. Resource-controlled remote execution to enhance wireless network applications. In *Proceedings of the 4th Workshop on Applications and Services in Wireless Networks*, Aug. 2004.

[18] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: a scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197. ACM Press, 1997.

[19] J. Nieh, C. Vaill, and H. Zhong. Virtual-time round-robin: An O(1) proportional share scheduler. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 245–259. USENIX Association, 2001.

[20] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. USENIX, 2002.

[21] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[22] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, 1995.

[23] R. Zhang, T. F. Abdelzaher, and J. A. Stankovic. Kernel support for open QoS-aware computing. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.