# Proportional-Share Scheduling: Implementation and Evaluation in a Widely-Deployed Operating System

David Petrou and John Milford

dpetrou@cs.cmu.edu, jwm@csua.berkeley.edu

*Draft 12/20/1997. Send feedback to the authors.*

**Abstract**

This paper explores the feasibility of using lottery scheduling, a proportional-share resource management algorithm, to schedule processes under the FreeBSD operating system. Proportional-share scheduling enables flexible control over relative process execution rates and processor load insulation among groups of processes. We show that a straight implementation of lottery scheduling performs worse than the standard FreeBSD scheduler. This initial result prompted us to extend lottery scheduling. Except for one test we run, our resulting system performs within one percent of the FreeBSD scheduler. We describe our design, evaluate our implementation, and relate our experience in deploying our lottery scheduler on production machines.

## 1 Introduction

This paper explores the feasibility of proportional-share resource management in a modern and well-understood operating system. Specifically, we employ lottery scheduling [Waldspurger & Weihl 1994] to allocate processor time in FreeBSD 2.2.5R. We describe and evaluate our implementation, a hybrid of the FreeBSD scheduler and lottery scheduling, and relate our experiences in using this scheduler on production machines. To begin, we motivate this paper with a summary of the benefits of proportional-share resource management.

Running processes make progress by consuming system resources such as processor cycles, network bandwidth, and storage. Considering processor cycles in particular, multiprogrammed systems allocate fixed quanta of processor time to running processes. A CPU scheduling algorithm based on proportional-share scheduling enables flexible control over the relative rates processes are allocated processor time quanta. For CPU-bound workloads, such as those found in some scientific and engineering environments, these rates are equal to the relative rates processes consume CPU.

For example, consider two independent CPU-intensive scientific simulations. These simulations begin running at timestep 0 and output intermediate results at the end of every timestep to be used by a 3-D rendering application. A researcher wants to visualize the results of these timesteps synchronously, but unfortunately the first simulation requires less computation and thus runs three times faster. Using a proportional-share process scheduler, the researcher can easily adjust the relative execution rates of the simulations so that the second process is chosen to run three times as often.

From a security perspective, proportional-share process scheduling enables users to carefully restrict the CPU consumption rate of untrusted binaries such as Java applications, or trusted binaries that use untrusted data such as WWW helper applications [Goldberg et al. 1996].

We have demonstrated the utility of proportional-share scheduling with respect to individual processes. Now we generalize and consider allocating proportions of processor time to individual users. In a time-sharing system, users own processes which compete for, among other resources, CPU time. Proportional-share process scheduling enables control over the relative rate at which users are permitted to use this resource. For example, one policy enforces users with CPU-bound jobs to consume the CPU at an equal rate. At a given time, if there is one user that wishes to make progress, he may consume 100% of the CPU, and if there are two users, they each may consume 50% of the CPU, *etc.*, regardless of the number of processes they own. Such control is useful for Internet service providers, in which hundreds of competing users log into one machine. With conventional processor schedulers, it is trivial for one user to monopolize the system with his own processes. Considering desktop workstations, another policy enables the console user to consume CPU time at a faster rate than users logged into the system remotely. Naturally, these policies also permit individual users to control the relative rate of CPU time that their own processes consume as in the scientific simulation example.

We believe that the features offered by proportional-share resource management are powerful and desirable. Lottery scheduling, introduced in 1994, is a scheduling algorithm that implements the described features but that is not in wide use today. Hence, our prime motivation was to determine if there are technical reasons why lottery scheduling has not supplanted standard operating system schedulers.

After building a straight implementation of the lottery scheduling algorithm we achieved the desirable features described. However, using the standard FreeBSD scheduler as a baseline for comparison, we experienced poorer responsiveness with interactive applications. We discuss the reasons for this unexpected performance anomaly and then present extensions to the lottery scheduling algorithm that remedy it. Our final implementation, which is currently running on two servers and one personal machine, nearly equals the performance of the FreeBSD scheduler under the benchmarks we run while providing the desired functionality of proportional-share resource management. These results indicate that lottery scheduling should be considered for wide use. However, we do not claim that our implementation makes "optimal" scheduling decisions. Our experience with implementing and examining a process scheduler have raised some interesting questions that we wish to explore in future work.

The rest of the paper is organized as follows. In section 2 we describe both FreeBSD's and the lottery scheduling algorithm. Section 3 explains our extensions to the core lottery scheduling algorithm while section 4 details our implementation. We evaluate and compare our lottery scheduler with the FreeBSD scheduler in section 5. In section 6 we present our experience in deploying our scheduler on production machines. Section 7 discusses several situations in which our scheduler "does the wrong thing" and in which correct solutions are not obvious. Finally, section 8 concludes our paper.

## 2 Background

An operating system's process scheduler has several conflicting goals. The scheduler must schedule interactive processes so that they are responsive to user input despite not being able to accurately distinguish interactive processes from non-interactive processes. The scheduler must also efficiently schedule batch processes to maximize throughput despite potential lock conflicts between such processes. At the same time, the scheduler must ensure that no processes starve. In some contexts, schedulers must ensure that processes meet real-time deadlines, although we are not concerned with such systems here.

Following, we contrast the characteristics and goals of the FreeBSD scheduler with the lottery scheduler.

### 2.1 Scheduling in FreeBSD

FreeBSD [Lehey 1996, FreeBSD 1997] is a UNIX [Ritchie & Thompson 1974] operating system for the Intel 80386 platform based on UC Berkeley's 4.4BSD-Lite [4.4BSD 1994, McKusick et al. 1996] release. FreeBSD's scheduler is a typical example of decay usage priority scheduling also used in System V and Mach with 100 ms time slices. The scheduler is implemented with a multi-level feedback queue. Processes with equal priority are placed onto the same queue. The scheduler runs round-robin the processes from the highest priority non-empty queue. The scheduler favors interactive processes via two mechanisms: (1) the scheduler lowers the priority of processes that consume CPU, thereby moving them to lower queues, and (2), the scheduler preempts processes before their quanta expire if a higher priority sleeping process wakes up. FreeBSD's scheduler also employs static priorities for processes holding kernel resources. These priorities are higher than priorities held by userlevel processes and exist to enable processes to release high-demand kernel resources quickly. Starvation is avoided by periodically raising the priority of processes that have not recently run.

The FreeBSD scheduler has several limitations. [Hellerstein 1993] demonstrates the difficulty in using priorities in decay usage schedulers to adjust processor consumption rates. It is non-trivial and potentially computationally expensive to dynamically adjust priorities of running processes to attain service rates such as those easily attainable from proportional-share schedulers. Further, FreeBSD attempts to provide load-insulation via two crude and unacceptable mechanisms. The operating system can limit the number of processes that one user may run simultaneously, and the operating system can terminate processes that accumulate more than a certain amount of processor time. These mechanisms prevent a user from starting many processes that consume processor time slowly and from completing a few processes that consume a lot of processor time over a long period of time. Lastly, we have noticed odd behavior under the FreeBSD scheduler when simultaneously starting a large number ($\approx$ 100) of CPU-bound processes. Initially, the responsiveness of the entire system (subjectively measured by observing typing and mouse movement latency) becomes very poor. After a few moments, the interactivity improves only to become poor again moments later. We believe this behavior results from the CPU-bound processes synchronously moving up and down in the priority queues.

### 2.2 Lottery Scheduling

Lottery scheduling is a simple scheduling algorithm that provides proportional-share resource management. Processes are assigned a number of *tickets*. When the scheduler needs to choose a process to run it performs a *lottery* and chooses the process that holds the winning ticket. The ratios of tickets that processes hold determines the ratios of expected times processes win lotteries. Only the processes that are runnable (not sleeping, stopped, swapped out, *etc.*) are eligible for the lottery. Starvation is avoided since no matter how few tickets a process holds, eventually it will win a lottery. We note that responsiveness with lottery scheduling appears to degrade more gracefully than with the FreeBSD scheduler when the system is loaded with a large number of CPU-bound processes.

A *currency* mechanism enables users and processes to hold tickets in different denominations, allowing the aggregate execution rate of all of a user's processes to be varied with respect to other users. This mechanism provides *load insulation* among users. When making scheduling decisions, the scheduler converts the tickets that a process holds into tickets in a *base* currency according to an exchange rate derived from the number of base tickets that fund a user's currency.

A *compensation tickets* mechanism improves the relative CPU consumption accuracy for preempted and short-sleeping processes and also improves responsiveness for interactive programs. A process that wins a lottery but only consumes a fraction of its time quantum is granted a temporary number of compensation tickets valid until the process is chosen to run again. The process holds an *effective* number of tickets, equal to the compensation tickets plus the process's tickets, during the next time the process participates in a lottery. For example, a process with 10 tickets that blocked after using half of its time quantum will receive 20 effective tickets during its next lottery, making it twice as likely to be chosen to run than without these compensation tickets. Ideally, processes that often block for very short periods of time (before the next scheduling decision is made) will receive processor time in proportion to the relative number of tickets that they hold regardless of what fractions of their quanta they use. Further, interactive processes, which generally block for a long period of time after using a small fraction of their quanta, are very likely to be chosen to run during the next scheduling decision after they wake up.

## 3 Design

The preceding section described our first lottery scheduling implementation following the description in [Waldspurger & Weihl 1994]. With this implementation we were able to control the relative execution rates of CPU-bound programs and insulate user workloads. In other areas the scheduler performed worse than the FreeBSD scheduler. We observed "choppy" response when editing files with background processes running and general system slowness when loading the system with many types of active processes. Our task at that point was to emulate features in the FreeBSD scheduler to remedy these anomalies while upholding the proportional processor utilization promises that lottery scheduling provides. In addition, user feedback prompted us to emulate the semantics of the UNIX `nice` utility. The following sections describe these topics.

### 3.1 Abbreviated Quanta

Users demand fast response from interactive applications. In the FreeBSD scheduler, various events on a process can potentially trigger a context switch before the running process uses its full time quantum. Two such events are a process waking up and a process receiving certain signals. If the process that receives the event has a higher priority than the running process, a context switch is forced immediately. The process will have a higher priority than the running process if it has used less CPU time than the running process recently (or if it was blocked in the kernel as described in section 3.2). We call this behavior *abbreviated quanta*, and we desire to replicate it in our lottery scheduler.

If a process receives an event of this type and has run for a shorter period of time during its last quantum than the currently running process[1], we force a context switch. The lottery scheduler will likely choose this process to run over other processes if it has earned many compensation tickets. To ensure that the preempted process will receive the processor in proportion to the tickets that it holds, the scheduler awards this process compensation tickets. (However, abbreviated quanta will still negatively affect processes that perform poorly when receiving less than a full time quantum, however). Note that after a process receives one of these events and we force a context switch, we do *not* unconditionally switch to this process but we rather perform the lottery as usual. One might therefore suppose that there is little utility to abbreviated quanta because the preempted process might simply reacquire the CPU. This chance exists, but it is not common, since we only force a context switch when the event-receiving process has used less CPU that the running process, making it probable that it earns more effective tickets than the preempted processes.

### 3.2 Kernel Priorities

Priority inversion [Lampson & Redell 1980, Birrell 1989, Hauser et al. 1993] is a well-documented problem in concurrent systems. Consider the following situation. A low priority process acquires a *non-preemptable* kernel resource such as a lock. A medium priority process begins running after a high priority process blocks waiting for the kernel lock to be released. The end-result is the high priority process remains blocked until the scheduler chooses the low priority process over the medium priority process.

In the FreeBSD scheduler, processes are assigned kernel priorities when they block waiting for a kernel resource. These kernel priorities are higher than priorities held by userlevel processes and are ordered in importance. Kernel priorities exist to enable processes to release high-demand resources quickly [Vahalia 1996, Jolitz & Jolitz 1996, McKusick et al. 1996].

The solution to priority inversion described in [Waldspurger & Weihl 1996] is for processes that are blocked on a resource to temporarily transfer their tickets to the process that holds the resource[2]. We admire the elegance of this solution, but it incurs overhead not found in the FreeBSD solution. A process that needs a kernel resources will find the resource in use, transfer its tickets, block, and trigger a context switch. These steps occur before the resource-holding process gets its boost in CPU consumption rate. The kernel priorities in FreeBSD minimize the chance that processes will find resources in use altogether.

Another example merits discussion. Consider a swapping process that goes to sleep waiting for a page from the swap device. In the FreeBSD scheduler, when the data becomes available the process wakes with a very high kernel priority, enabling it to be chosen to run during the next context switch. In this example, the resource held by the process when it wakes up, a page in main memory, is *preemptable* by other processes. However, preempting this resource is very expensive. A lottery scheduler that ignores kernel priorities can exhibit the following pathological behavior. A process wakes up when it receives a needed page. The process, however, is not chosen to run during the next context switch because there are other processes with more effective tickets in the base currency. These other processes, also needing memory, cause either the recently loaded page or another page within the process's working set to be evicted, requiring another page fault before the process can continue. The end-result is poor system throughput.

These are examples of the straight lottery scheduling algorithm proportionally provisioning the processor, yet failing to provide good overall system performance. The underlying reason why lottery scheduling fails is because it makes localized resource allocation decisions without considering the behavior of the system as a whole. We remind the reader that our goal was to achieve performance similar to the FreeBSD scheduler. To that end, we implemented the following kernel priority extension to lottery scheduling.

We maintain a list of processes that have woken up after being blocked in the kernel sorted by the FreeBSD kernel priority number. Before performing the lottery during a context switch, the scheduler checks to see if the list of processes with kernel priorities is empty. If it is not empty, the scheduler chooses the first process on this sorted list to run. At this point we solve the problems described above but we violate the proportional processor allocation that was the motivating reason for implementing lottery scheduling.

We now describe our fix that gives us the best of both worlds. A timer tracks the total amount of time that the process has run for, starting when it was picked in a lottery until it is descheduled in userland. The process may be descheduled many times in the kernel as it acquires kernel resources and the timer continues to increase when the process runs until the process is descheduled outside of the kernel. At this point, the scheduler computes its compensation tickets as described in section 2.2. If the process used more time than one time quantum, a *negative* number of

[1] We actually want to compare the number of effective tickets in the base currency that the event-receiving process holds with the effective tickets in the base currency that would be earned by the running process if it is preempted at this point. These values are expensive to compute so we use the time heuristic instead. If our heuristic fails and we unnecessarily force a rescheduling event, our scheduler still performs correctly albeit there is the added overhead of an extra context switch.

[2] This mechanism is similar to *priority inheritance* but permits additive ticket transfers from multiple blocked processes.

compensation tickets results. The more a process overran its quantum, the more negative compensation tickets it receives. Negative compensation tickets make a process *less* likely to be chosen to run. Our timer ensures that we do not violate the lottery scheduling goals[3] while enabling the scheduler to reduce kernel resource contention.[4]

### 3.3 `nice` Emulation

While deploying our lottery scheduler to production systems we received requests for our scheduler to support the UNIX `nice` utility. `nice` enables a user to vary the priority of his processes from -20 (high) to 20 (low) relative to all other processes in the system. We first show that mapping `nice` semantics to the lottery scheduler is not obvious and then present our design.

A naïve approach to emulating `nice` semantics is to map nice values to a process's tickets. This approach fails because only the relative execution rate of a user's own processes are affected. If a user has only one CPU-bound job running that is `nice`'d to +20, then it will receive few tickets, but since it is the only running process belonging to that user, the exchange rate will give it all of the base tickets that fund that user's currency. Alternatively, `nice` can affect the base tickets that fund a user instead of affecting the process's tickets. This approach also fails, but for a different reason. Now if a user `nice`'s a process, all of the user's processes will have a lower priority.

We implement the following solution. We map `nice` values to tickets and base tickets. When a process is positively `nice`'d, the process holds *at most* a number of tickets equal to the `nice` value to ticket mapping and *at most* a number of base tickets equal to the `nice` value to base ticket mapping when a scheduling decision is made. When a process is negatively `nice`'d, the process holds *at least* a number of tickets and *at least* a number of base tickets equal to the respective `nice` value mappings when a scheduling decision is made. (Recall that only root can negatively `nice` a process.) This solution avoids the problems with our approaches described above while satisfactorily emulating `nice` semantics and remaining in line with the lottery scheduler's goals.

## 4 Implementation

Our system is divided into two parts. The bulk of the code is in the kernel files implementing our lottery scheduler. The rest of the system are small userlevel programs that make system calls to adjust scheduling parameters. For brevity, we leave out some implementation details.

[3] There is a slight complication to our algorithm. Consider a process that spends very little time in userland because it continually makes blocking system calls. Such a process will receive the processor during the next context switch after it wakes up in virtue of having a kernel priority. Eventually it will be descheduled in userland (by the time slice interrupt) and it will receive a large number of negative compensation tickets, causing it to wait a relatively long time before running again. We would prefer a process to lose its kernel priority status more often and incur fewer negative compensation tickets. Our solution is to force a context switch if the process is entering userland (implying that it does not hold a kernel priority) and has accumulated more than one time quanta on its timer.

[4] It may be possible to achieve higher process throughput in the kernel if the ordering of FreeBSD kernel priorities is incorrect for certain workloads. However, as our goal is to achieve performance similar to FreeBSD's scheduler, we do not attempt this.

### 4.1 Kernel Functionality

To describe our kernel code we first describe critical pieces of the standard FreeBSD scheduler. There are 32 multi-level feedback queues called *runqueues*. After becoming runnable, processes are put onto the appropriate runqueue by the assembly language routine *setrunqueue()*. Processes are removed from runqueues when chosen to run by the scheduler. The assembly language routine, *cpu_switch()*, (1) saves process context, (2) chooses the next process to run (if no processes are runnable it idles), and (3) switches to that process. Apparently for performance considerations, this function violates the good programming practice that functions should "do one thing well."[5] [Lampson 1984]

We modified *cpu_switch()* to decouple the scheduling decision from the scheduling mechanism. Specifically, we rearrange some of the assembly code and make calls to a C function called *lott_choose_next_runner()*. We store ticket information along with other necessary scheduling information in a per-process structure called *lott_proc*. During boot-time initialization, our function *lott_init()* initializes a hash table that allows us to access *lott_proc*'s quickly. We also maintain a doubly-linked list of processes that are runnable. We replace *setrunqueue()* with a C function that manipulates this list. We add the "hand-created" process 0 and all future processes created by *fork1()* to our hash table with a call to *lott_add_proc()*, which assigns processes an initial 10 tickets. Likewise, processes are removed in *exit1()* with a call to *lott_remove_proc()*. We use a hash table to efficiently access a data structure called *lott_user* which holds the number of base tickets that fund all of the runnable processes corresponding to a user ID and a reference count that enables us to garbage collect these structures when all of the processes owned by a user terminate. The *setuid()* system call (*cf.* `login`) is modified to call *lott_add_user()* which creates a *lott_user* structure and funds the user with 1000 base tickets.

Any lottery scheduling implementation with compensation tickets and currencies will be more computationally expensive than the FreeBSD scheduler. As our goal was to achieve comparable performance to FreeBSD, we expended much effort in optimizing our implementation. Motivated by [Massalin & Pu 1989] we factor invariants, defer work, use many inline functions, and aggressively cache computed values.

The heart of the lottery scheduling algorithm is in *lott_choose_next_runner()*. First we check to see if both the lists of processes with kernel priorities and without kernel priorities are empty. If they are empty, we return 0, which causes *cpu_switch()* to idle. If the kernel priority list is not empty, we select the first process on this list to run. If it is empty, we proceed to the lottery scheduling algorithm. We need to find the total number of effective tickets in the system so that we can choose a winning ticket within this range. For each runnable process we perform the following steps. We check to see if our cached value of the process's effective tickets is valid. If it is not, we compute its effective tickets by dividing the number of microseconds in one time quanta by the number of microseconds used by the process (this value is stored by the timer described in section 3.2). We

[5] The opaque structure of this function is the reason for, in our opinion, it remaining largely unmodified from the earliest version of the code that we found. [Parnas 1972] explains that systems should be decomposed based on "difficult design decisions or design decisions which are likely to change." We find the poor modularization of the FreeBSD scheduler unfortunate, as it impedes scheduler experimentation.

then multiply this value by the number of tickets that the process holds to get the number of effective tickets. Now we check to see if our cache of the exchange rate between the process's effective tickets and tickets in the base currency is valid. If it is not, we compute this exchange rate by dividing the number of base tickets that fund the user's currency by the number of tickets in all of the user's runnable processes. We then check to see if our cache of base tickets held by the process is valid. If it is not, we compute it by multiplying the user's exchange rate by the process's effective tickets. At this point we implement the nice "at most/at least" algorithm described in section 3.3 by ensuring that the number of base tickets lies within the allowable range[6]. We go through these computations for each process and maintain a running count of the total number of base tickets in the system. When finished, we pick a random number between one and this number. Finally, we iterate through the list of runnable processes one more time to find which process holds the winning ticket.

A few details are worth mentioning. Floating point operations are not permitted in the kernel, so for accuracy we use fixed point arithmetic. Different parts of the algorithm require different amounts of precision, and some variables are capable only of representing a small amount of precision because the rest of their bits are required to represent whole number components. Therefore we use both 8 and 12 bit fractional components for different parts of the algorithm. For most of the algorithm we compute with 32-bit integers, allowing us to do arithmetic purely in hardware. Toward the end of the computation we must use 64 bits. Specifically, when computing a process's base tickets, the exchange rate and the effective tickets are both 32-bit values that when multiplied, are likely to result in a 64-bit integer. The Intel 80386 architecture can do a 32-bit times 32-bit to 64-bit operation in hardware. We use an inline assembly instruction for this operation as we could only get the gcc compiler to either store the result as a 32-bit integer (losing the most significant 32 bits) or promote the operands and perform a 64-bit times 64-bit to 64-bit slow software multiplication. The only 64-bit software emulated arithmetic in our algorithm is where we mod the random number by the total number of base tickets in the system.

We instrumented both the unmodified FreeBSD kernel and our lottery scheduler to provide us with profiling information, some of which will be discussed in sections 5 and 6.2. We can display information about the last $n$ scheduling events, including information about processes releasing the CPU, processes waking up, and time quanta expirations. This information has been invaluable in helping us debug our implementation and in measuring our system against the FreeBSD baseline. A user program requests this information by making a system call with arguments that specify a buffer location in userspace, and the number, $n$, of entries requested. The user process then blocks until $n$ scheduling events are made. At this point, the kernel calls *copyout()* to copy the requested information to the user buffer and calls *wakeup()* to unblock the user process. We note that there were several potential synchronization problems including the "lost-wakeup" problem that we had to avoid when implementing our profiling code.

[6] The mapping from nice values to base tickets is basetickets = $10^{\frac{1}{20}(-\texttt{nice}+20)+2}$.

### 4.2 Userlevel Programs

We now show how users adjust lottery scheduling parameters. All of these programs make system calls to achieve the described functionality. The first set of functions are available to all users.

set_tickets -p<pid> -t<tickets> — Changes the number of tickets held by <pid> to <tickets>. The user must own <pid> unless the user is root. <tickets> is in user tickets (as opposed to base tickets).

run_tickets <tickets> <prog> [<arg1> ...] — Executes <prog> with optional <argN>'s using <tickets> represented in user tickets.

show_tickets [-u<uid>] [-p<pid>] [-x] — Shows the number of tickets held by <pid>. The <pid> must be owned by the user, unless the user is root. A <pid> of -1 shows the number of tickets held by all of the user's processes, the default. Root can view information on other users by specifying the <uid>, or -1 for all users. -x shows additional information.

set_funding [-u<uid>] -t<base_tickets> — Root sets the number of <base_tickets> that funds <uid>'s processes. The default <uid> is the user ID running the program.

show_funding [-u<uid>] — Shows the number of base tickets that fund <uid>. The default <uid> is the user ID running the program. If <uid> is -1, all users are shown. Only root can view another <uid>.

force_sched -p<pid> -m|-l|-n <base_tickets> — Enables a user to control the "at most/at least" nice emulation without explicitly making the *setpriority()* system call used by nice and renice. -p chooses a process. -m means schedule the process with at most <base_tickets>. -l means schedule the process with at least <base_tickets> (restricted to root). -n means use standard lottery scheduling.

lott_chuser -p<pid> -u<uid> — Takes <pid> and puts it under <uid>'s currency. Useful for moving processes like X which run as root under the currency of the user using the process. This command is restricted to root.

The following programs are available only for lottery scheduling development.

lott_ctl -a <on|off> — Root-only command that turns on or off the abbreviated quanta feature. Future controllable features will be turned on and off via this command.

lott_stat [-i<on|off>] [-l<on|off>] [-c<on|off>] [-p<pid>] — Turns on or off profiling information gathering and printing. -i controls statistical information printed out to the system console in real-time. -l controls recording of process scheduling data retrievable via lott_rinfo. -c controls recording of cycle count performance data for scheduling functions, again retrievable via lott_rinfo. The defaults for these options are off. -p makes the output reflect only the chosen process, -1 for all processes, the default.

lott_rinfo [-p<on|off>] [-i<on|off>] [-q<on|off>] [-c<on|off>] [-s<on|off>] -n<num_entries> — Outputs data concerning the next <num_entries> scheduling events as described in section 4.1. -p controls reporting of process information. -i controls reporting of when the kernel is idling. -q controls reporting of when time slices expire. -c controls reporting of cycle count performance data. -s alters the output to be machine readable for statistical analysis. All of the previous options are on by default except for -s.

## 5 Evaluation

The variety of potential workloads makes evaluating scheduler performance difficult. In general, however, our day-to-day experience matches the results that we obtain from the benchmarks in this section. Exceptions are explored in section 6.

All experimental results were obtained from partita, David Petrou's personal machine. This machine has 64 MB of main memory and a 200 MHz AMD K6 (pentium compatible) processor. Unless otherwise noted, no tests caused the machine to page. In the following figures error bars represent 95% confidence intervals.

We first show that the overhead incurred by our lottery scheduler is minimal, and then we demonstrate the proportional-share resource management properties that we gain with the lottery scheduler.

### 5.1 Overhead

We divide our overhead measurements in two parts: (1) microbenchmarks that quantify the differences in time it takes to execute scheduling operations under both the FreeBSD scheduler and the lottery scheduler kernels, and (2) macrobenchmarks that determine whether these differences are visible when running applications.

#### 5.1.1 Microbenchmarks

We measure scheduling code fragments to quantify time spent in scheduler overhead. We use the the cycle counter instruction `RDTSC` (Read Time-Stamp Counter) which increments a counter every clock cycle [Int 1996] to obtain accurate measurements. As we ran our tests on a 200 MHz machine, divide "cycles" by 200 to obtain microseconds.

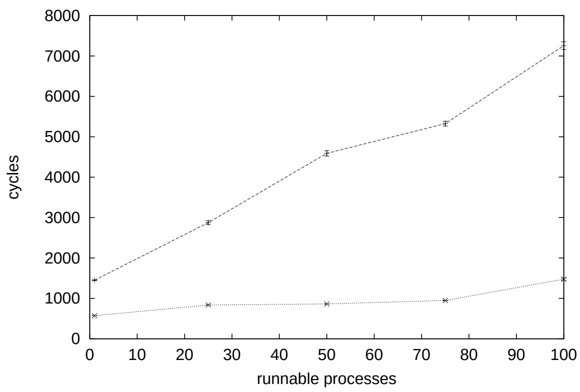


Figure 1: This figure shows the average number of cycles (out of at least 1000 measurements) to perform a context switch via the *cpu_switch()* function while varying the number of runnable processes. The top curve represents lottery scheduling while the bottom represents the FreeBSD scheduler.

The two most common scheduling operations in both the FreeBSD and our lottery schedulers are *cpu_switch()* and *setrunqueue()*. Under the FreeBSD scheduler, *cpu_switch()* makes a scheduling decision and performs a context switch. In our lottery scheduler, *cpu_switch()* calls *lott_choose_next_runner()* to make a scheduling decision and performs a context switch. Figure 1 shows the number of cycles it takes to run *cpu_switch()* while varying the number of processes[7]. Our lottery scheduling algorithm is O($n$) in the number of runnable processes while the FreeBSD scheduler is O(1). This difference in algorithmic complexity is visible in these results. Figure 2 shows the number of cycles to execute *setrunqueue()*, which makes a process runnable. This function is short and O(1) in both schedulers. We believe that the FreeBSD version is faster because it is written in assembly. We do not know why the FreeBSD curve fluctuates. With small standard errors, those fluctuations are statistically significant. Table 1 presents this data in numerical format.

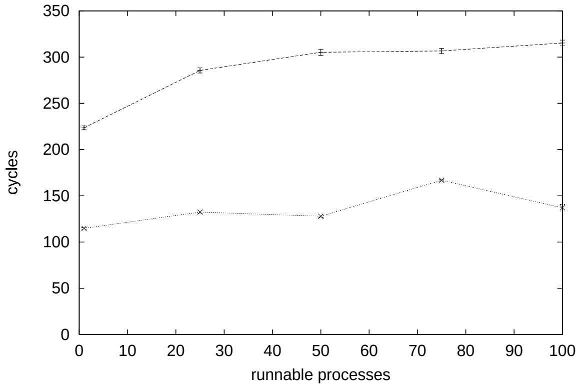


Figure 2: This figure shows the average number of cycles (out of at least 1000 measurements) to mark a process runnable via the *setrunqueue()* function while varying the number of runnable processes. The top curve represents lottery scheduling while the bottom represents the FreeBSD scheduler.

#### 5.1.2 Macrobenchmarks

The previous section uncovered measurable differences between the FreeBSD scheduler and our lottery scheduler. In this section we determine how visible these differences are on a larger scale. We measure two classes of programs: (1) interactive programs, and (2) batch programs. Our quality of service metric for interactive applications is response time while we consider throughput for batch applications.

We examine interactive applications first. We wrote a benchmark called `interactive` that continually goes to sleep for the shortest time possible and measures the time between going to sleep and the time that it receives the processor when it wakes up. While we run `interactive`, we also run a CPU-bound process in the background. Again, we use the `RDTSC` cycle count instruction. We keep in mind that the minimum latency that humans can discern varies between 50–150 ms depending on the individual [Shneiderman 1992]. Figure 3 measures response time under the FreeBSD scheduler as a baseline. Due to the way timeouts are handled in FreeBSD, processes cannot sleep on a timer event for less than 20 ms. Figure 4 shows the same experiment under the lottery scheduler. We achieve very similar results. We were curious about the utility of abbreviated quanta, so we disabled this feature within our lottery scheduler and reran this test. Our poor results are presented in figure 5. Table 2 presents the data in these figures numerically.

We now consider throughput of batch processes. Our test application is `rc564` [rc564 1997], a program that tries to find the solution to RSA's 64-bit secret-key challenge. To exacerbate the affect of our added overhead while running `rc564`, we also run a varying number of `interactive` processes. These `interactive` processes, due to abbreviated

[7] Naturally, we include the cycles in *lott_choose_next_runner()* in these measurements.

| | | FreeBSD Scheduler mean/std. err. | Lottery Scheduler mean/std. err. |
|---|---|---|---|
| 1 process | *cpu_switch()* | 572.54/2.17 | 1449.13/3.74 |
| | *setrunqueue()* | 114.79/0.51 | 223.62/1.06 |
| 25 processes | *cpu_switch()* | 835.90/2.35 | 2874.79/24.76 |
| | *setrunqueue()* | 132.40/0.67 | 285.59/56.73 |
| 50 processes | *cpu_switch()* | 863.11/2.46 | 4589.64/34.30 |
| | *setrunqueue()* | 127.83/0.74 | 305.19/1.70 |
| 75 processes | *cpu_switch()* | 948.68/2.84 | 5325.49/30.55 |
| | *setrunqueue()* | 166.88/0.53 | 306.70/1.40 |
| 100 processes | *cpu_switch()* | 1474.50/13.14 | 7255.03/48.29 |
| | *setrunqueue()* | 137.00/1.78 | 315.27/1.58 |

Table 1: This table presents the data from figures 1 and 2 in numerical format. The values are in number of cycles.

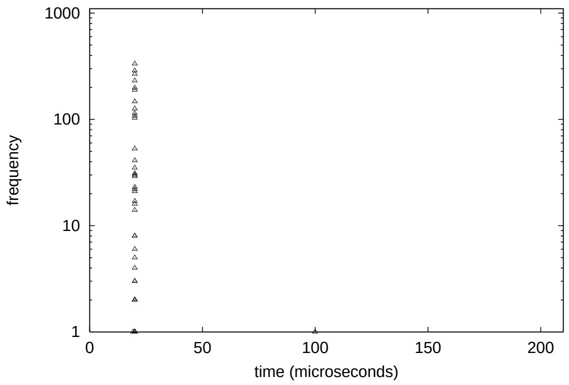


Figure 3: This histogram shows the response time distribution of `interactive` under the FreeBSD scheduler while a CPU-bound process runs. Almost all times are clustered around 20 ms.

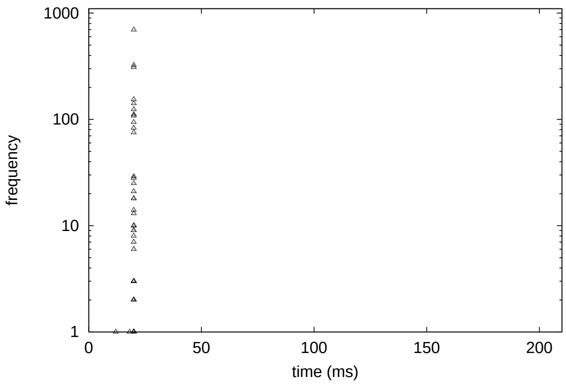


Figure 4: This histogram shows the response time distribution of `interactive` under the lottery scheduler while a CPU-bound process runs. As in the FreeBSD scheduler test, almost all times are clustered around 20 ms.

| | mean/std. dev. |
|---|---|
| FreeBSD Scheduler | 20.02/1.60 |
| Lottery Scheduler | 20.03/2.00 |
| Lottery (no a.q.) | 100.39/7.73 |

Table 2: This table presents the data from figures 3, 4, and 5 in numerical format. The times are in milliseconds. Over 2000 numbers were gathered for each case.

quanta, increase the number of context switch operations as shown in figure 6. The throughput of `rc564` versus the number of `interactive` processes is shown in figure 7. We note that as more `interactive` processes are run, the performance of `rc564` under the FreeBSD scheduler and the lottery scheduler worsens and diverges. In all runs, `rc564` under the lottery scheduler is less than one percent slower than under the FreeBSD scheduler.

As the previous experiment did not show a large difference between the FreeBSD and lottery scheduler, we ran one last macrobenchmark. In figure 8 we show the progress of a program called `count`, which simply loops and maintains a counter of how many loops it made, while running 100 `interactive` processes. These `interactive` processes pushed the number of context switches per second up to 5160 averaged over the run. Again, we achieve performance within one percent of the FreeBSD scheduler. One potential criticism is that there are systems that context switch more than 5160 times per second which may cause our scheduling overhead to be more apparent. However, we measured the average number of context switches per second over a 30 second interval on `wcarchive`, the world's largest and busiest FTP site,[8] to be 2589.

### 5.2 Flexible Execution Rate Control

The previous section demonstrated that the overhead of our lottery scheduler is negligible for the tests that we ran. Now we demonstrate the features that we have gained by replacing FreeBSD's decay usage scheduler with lottery scheduling. These are simple measurements and we refer the reader to [Waldspurger 1995] for an extensive analysis of lottery scheduling.

We demonstrate the ease at which a user can control the execution rate of his programs in figure 9. This figure shows three processes assigned tickets in a 3:2:1 ratio. Their

[8] `wcarchive` supports up to and often reaches 2750 simultaneous connections and stores 142 GB on-line. `wcarchive` is located at ftp://ftp.cdrom.com/.

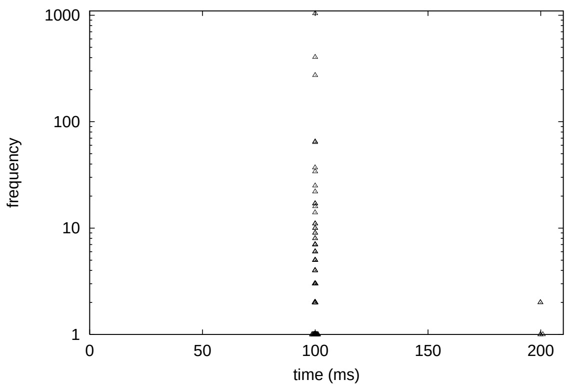


Figure 5: This histogram shows the response time distribution of `interactive` under the lottery scheduler without our abbreviated quanta extension while a CPU-bound process runs. The process now must wait a full time quantum, 100 ms, before running.

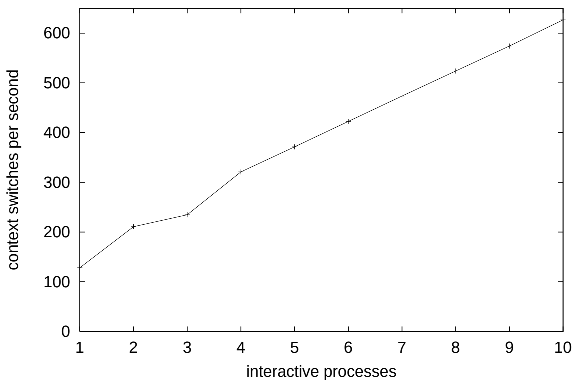


Figure 6: This figure shows the effect of increasing the number of `interactive` processes on the number of context switches per second.

progress is plotted along with ideal lines. We were curious how hard this was to reproduce using the FreeBSD scheduler. We achieve something close in figure 10. However, the `nice` values that we discovered, 10, 5, and 0, are hardly intuitively mappable to our goal of 3:2:1. The curves are also somewhat asymptotic and may diverge over time. Further, while lottery scheduling maintains the 3:2:1 ratio irrespective of system load, the FreeBSD scheduler unpredictably schedules these processes if the system is otherwise loaded.

We demonstrate user workload insulation in figure 11. Despite one user running two CPU-bound processes, the second user is able to receive twice the throughput from his one CPU-bound process. We also demonstrate user workload insulation with respect to processes that consume CPU and I/O. Figure 3 shows the time it takes to compile a program under an unloaded FreeBSD scheduler, under the FreeBSD scheduler with another user running 10 CPU-bound processes, and under the lottery scheduler with the same 10 process load. With lottery scheduling, the compiling user makes

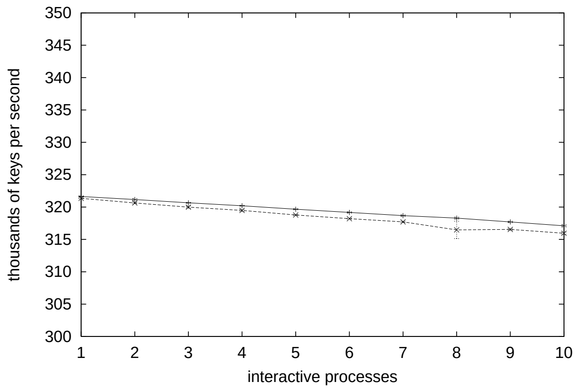


Figure 7: This figure shows the number of keys tried by `rc564` per second while varying the number of interactive processes. The top curve represents throughput under the FreeBSD scheduler while the bottom represents the lottery scheduler. Notice that the scale on the y-axis does not begin at zero. The performance of the system under the lottery scheduler is within 1% of the FreeBSD scheduler.

progress much faster than under the FreeBSD scheduler[9]. One may doubt the likelihood of a system with 10 runnable processes. A simple count showed 167 runnable processes (out of 2820 total processes) on `wcarchive`.

| | mean/std. err. |
|---|---|
| Idle FreeBSD Scheduler | 4.95/0.04 |
| Loaded FreeBSD Scheduler | 48.55/1.52 |
| Loaded Lottery Scheduler | 16.11/0.19 |

Table 3: This table shows the average time in seconds of 10 trials to compile and link a 2275 line program. We run this test under an unloaded FreeBSD scheduler, the FreeBSD scheduler while another user runs 10 CPU-bound jobs, and the lottery scheduler while another user runs 10 CPU-bound jobs.

## 6 Experience

Here we discuss our experience in using the lottery scheduler on two production machines and one personal machine.

### 6.1 `soda` and `meeko`

We have deployed our lottery scheduler on two production machines, `soda.csua.berkeley.edu` and `meeko.eecs.berkeley.edu`. `soda` is the central machine for the Computer Science Undergraduate Association at UC Berkeley. `soda` has 2300 accounts and often has over one hundred users logged on. Common users activity include participation in a chat room and code development. `soda` also manages 973 mailing lists besides serving mail for its users. `meeko` belongs to the FreeBSD Users' Group at UC Berkeley. `meeko` offers WWW service and mirrors part of `wcarchive` on its FTP

[9] We expected the compile under lottery scheduling to run faster than shown. We are investigating what our bottleneck is.

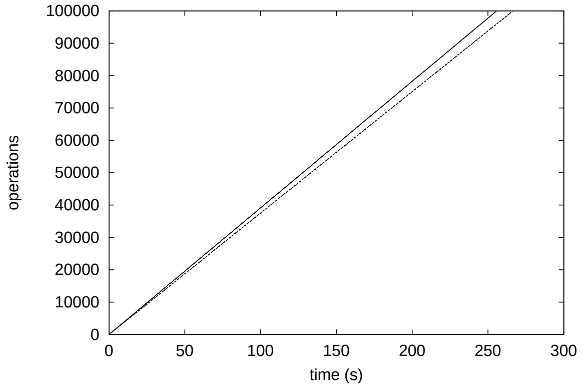


Figure 8: We run `interactive` 100 times and plot the progress of `count`. The FreeBSD scheduler is represented by the curve with the steeper slope while the second line represents progress under the lottery scheduler. The `interactive` processes caused 5160 context switches on average, well above what we experience on the busiest systems. Lottery scheduling is still less than 1% slower in this case.

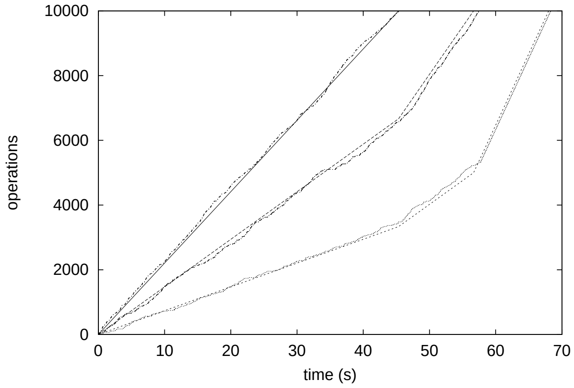


Figure 9: This figures demonstrates the progress of three `count` processes under lottery scheduling. The curves from top to bottom have 30, 20, and 10 tickets respectively. Also shows are straight lines representing ideal processor utilization. Notice that when processes finish, the remaining processes execute faster.

server. In addition, `meeko` exports a filesystem via NFS. There are usually 5 users logged into `meeko` actively developing code. That these systems have been running and in heavy use for weeks and that we have received no performance complaints from users is a testament to our code's stability and performance. Users report no lag related to the scheduler showing that the compensation ticket mechanism coupled with abbreviated quanta and kernel priorities are sufficient to provide good responsiveness.

We determine that load insulation works on our deployed code by looking at the output of the UNIX `top` utility while two users run the CPU-bound processes `xoopic` and `rc564`. `xoopic` [xoopic 1997] is a particle-in-cell plasma simulation that calculates particle positions and velocities by discretizing Maxwell's equations in time and space on a 2-D mesh. Tables 4 and 5 show the results.

| PID | USERNAME | PRI | NICE | SIZE | RES | STAT | TIME | WCPU | CPU | COMMND |
|---|---|---|---|---|---|---|---|---|---|---|
| 555 | jwm | 92 | 0 | 808K | 164K | RUN | 0:17 | 16.34% | 16.25% | rc564 |
| 553 | peterm | 90 | 0 | 7392K | 8012K | RUN | 0:18 | 16.28% | 16.21% | xoopic |
| 552 | peterm | 90 | 0 | 7392K | 8012K | RUN | 0:18 | 16.12% | 16.06% | xoopic |
| 550 | peterm | 90 | 0 | 7392K | 7852K | RUN | 0:18 | 16.12% | 16.06% | xoopic |
| 551 | peterm | 90 | 0 | 7392K | 7864K | RUN | 0:18 | 16.08% | 16.02% | xoopic |
| 554 | peterm | 89 | 0 | 7392K | 8012K | RUN | 0:18 | 16.05% | 15.98% | xoopic |

Table 4: This table shows output from `top` while two users are running one and five CPU-bound processes respectively under the FreeBSD scheduler. The lack of load insulation enables `peterm` to obtain an unfair percentage of the CPU.

We have experienced an interesting problem with our emulation of `nice` semantics on `meeko`. Memory was slightly overcommitted and we `nice`'d `rc564` and `interactive` processes to -20 (highest priority). At this point the performance of many processes plummeted and a lot of CPU time was spent in the kernel (system time). Our untested theory is that other processes were spending most of their time page faulting. Before one of these low priority processes is able to make progress in userland it is context switched out

| PID | USERNAME | PRI | NICE | SIZE | RES | STAT | TIME | WCPU | CPU | COMMND |
|---|---|---|---|---|---|---|---|---|---|---|
| 296 | jwm | 98 | 0 | 808K | 392K | RUN | 0:28 | 52.21% | 48.71% | rc564 |
| 272 | peterm | 76 | 0 | 7392K | 7544K | RUN | 1:02 | 11.63% | 11.63% | xoopic |
| 275 | peterm | 65 | 0 | 7392K | 7716K | RUN | 0:57 | 9.61% | 9.61% | xoopic |
| 282 | peterm | 64 | 0 | 7392K | 8032K | RUN | 0:50 | 9.50% | 9.50% | xoopic |
| 274 | peterm | 55 | 0 | 7392K | 7636K | RUN | 0:57 | 7.90% | 7.90% | xoopic |
| 273 | peterm | 53 | 0 | 7392K | 7600K | RUN | 0:55 | 7.13% | 7.13% | xoopic |

Table 5: This table shows output from `top` while two users are running one and five CPU-bound processes respectively under the lottery scheduler. `jwm` is able to receive about 50% of the CPU despite having only one runnable process.

and stands the chance of having pages in its working set evicted. We currently map a nice value of -20 to 10000 base tickets, an order of magnitude higher than the default number of base tickets, 1000, that fund a user's currency. We believe that this mapping is too aggressive and the problem we observed may diminish if we map -20 to a lower number such as 5000 base tickets.

### 6.2 `partita`

`partita` is the only machine running our lottery scheduler that is also running the X window system. We have observed the following problem that we have not been able to address. We experience slight choppiness when holding a key down (30 keys/s) in `emacs` using X while also running a CPU-bound process in the background. After closely examining the output from our `lott_rinfo` scheduler profiler, we determined that our scheduler was functioning correctly, and that `emacs` was running as soon as it had a keystroke, preempting the CPU-bound job. We also found no choppiness when running `emacs` outside of X. From the following series of events, we see that it is X that is not running as often as it should, causing the perceived choppiness: First X is running and `emacs` is asleep waiting for a keystroke. We press a key. Since we implement abbreviated quanta, and since `emacs` holds a kernel priority, `emacs` preempts X and runs immediately. Now `emacs` goes to sleep waiting for

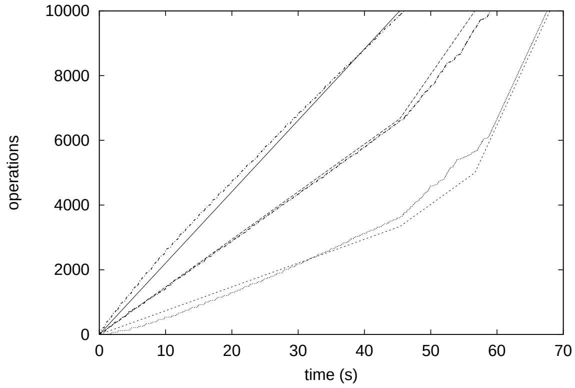


Figure 10: This figures demonstrates the progress of three `count` processes under the FreeBSD scheduler. We sought to emulate figure 9 and ran the programs from top to bottom with `nice 0`, `nice +5`, and `nice +10` respectively. Also shows are straight lines representing ideal processor utilization.

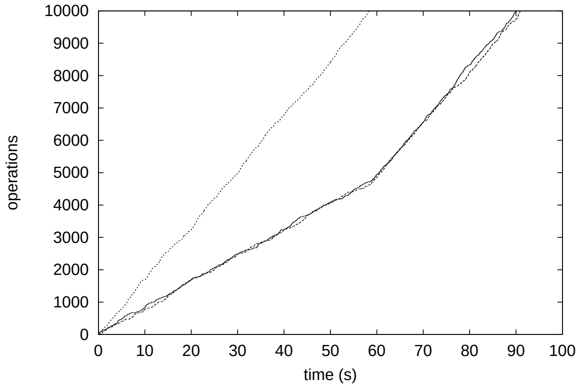


Figure 11: This figure shows the progress of three `count` processes under lottery scheduling. The top curve represents a `count` process run by one user while the bottom curves represent two `count` processes run by another user. When the first `count` finishes, the other `count` processes have made 47% and 46% progress toward completion (50% is ideal).

the next keystroke. Up to this point, both the FreeBSD scheduler and the lottery scheduler behave the same. Both X and the CPU-bound job are runnable and as both processes were preempted running userlevel code, neither have a kernel priority. X probably has more compensation tickets than the CPU-bound job, making it more likely to be chosen to run, but occasionally, the CPU-bound job will be run instead, resulting in choppiness. In the FreeBSD scheduler, the CPU-bound job runs often enough during the times when both X and `emacs` are sleeping to attain a lower priority. In this case, X always holds a higher priority than the CPU-bound job and is always chosen to run first. We can virtually eliminate this choppiness under the lottery scheduler by assigning the X window process 100 tickets, ten times more than the default of 10 tickets. We are trying to determine how common this problem occurs while investigating more general solutions.

## 7 Future Work

Besides the X anomaly described in the previous section, our work raises many issues that we leave for future work. We describe some of these in turn.

The ticket transfer mechanism is one aspect of lottery scheduling that we have not implemented. If one process desires a result that another process computes, the first process can "loan" its own CPU consumption rate to the second process. Now this second process will use CPU time at the aggregate rate initially granted to both processes. When the result is computed, it sends the result to the first process along with the right to consume CPU at the certain rate "borrowed" from the first process.

For example, a process blocked on interprocess communication can loan its tickets the server processes. Clients and servers can be explicitly written to make these transfers, or we can try to provide this functionality automatically. We notice that most IPC take the form of *read()* and *write()* system calls. In many cases we can determine which processes are communicating and automatically transfer tickets from the reader (the client blocked on a result) to the future writer (the server computing the result), and back again after the write completes. Such transfers can alleviate priority inversion among userlevel code.

In a distributed setting, an RPC layer can be written to transfer tickets among clients and servers. Consider a busy web server. Important clients can pay money for a certain number of tickets. These tickets are transmitted to web servers along with a URL in an unforgeable capability. The web server then schedules responses to queries based on the number of tickets that web clients hold.

There are some client/server scenarios on a single machine in which it is not clear that ticket transfers will work. Consider the printer command `lpr` and the printer daemon `lpd` owned by root. A user executes `lpr` which terminates and then at some time in the future, `lpd` executes on that user's behalf. This violates load insulation as the root-owned process is working on behalf of a user from its own tickets. There is no clear way to transfer tickets from the user to root because `lpr` has terminated before `lpd` runs. Similar arguments can be made for other system daemons such as `sendmail`.

A more difficult problem concerns scheduling diverse resources. For example, consider the `pagedaemon` process, a root-owned process that moves pages between memory and the swap device on behalf of all users. A process that is paging is not penalized for the time that the `pagedaemon` is executing on its behalf. On one hand, we would like to reduce the rate at which large processes page memory on the assumption that this will free up disk-bandwidth for other processes. Unfortunately, this may simply cause the large process to page even more as it doesn't have a chance to use its pages before they are evicted.

[Waldspurger & Weihl 1994] explains that proportional-share resource management can be applied across diverse resources but does not provide an algorithm or policy by which an entire system can be scheduled toward optimal system performance. [Hauser et al. 1993, Nieh et al. 1994] suggest that adjusting priorities (tickets) toward higher-level scheduling goals is very difficult or intractable. We find the pursuit of more intelligent system-wide resource schedulers an exciting research question. We envision a framework by which a process registers its scheduling goals and the operating system schedules system resources to the process based on ticket-like objects. A cost-benefit analysis such as in [Patterson et al. 1995] considers the types of resources requested and how other processes may be affected by having those resources revoked.

## 8 Conclusion

We began this work with the goal of discovering why lottery scheduling, and by extension proportional-share scheduling, is not a standard part of modern operating systems. Our initial implementation followed [Waldspurger & Weihl 1994] and enabled control over process execution rates and processor load insulation at the cost of system responsiveness. After examining the FreeBSD scheduler, we decided to apply both abbreviated quanta and kernel priorities to our lottery scheduler. These techniques have been applied without altering the proportional-resource management semantics. In addition, user feedback prompted us to add support for the UNIX `nice` utility. Our measurements show that our optimized scheduler incurs more overhead than the FreeBSD scheduler, but that these differences are negligible even under large workloads. We achieve throughput and responsiveness nearly equal to FreeBSD except for one case concerning the X window system. Our remaining benchmarks indicate that we do achieve flexible control over the rate at which processes consume CPU. Our lottery scheduler was deployed to two production machines where we observe similar results. Our experience spurred many issues that we wish to explore in future work. This paper demonstrates that lottery scheduling is a viable process scheduler for the workloads we have tested. Our findings warrant further investigation into incorporating lottery scheduling as a standard part of operating system kernels.

## 9 Availability

Our lottery scheduler is available from http://www.cs.cmu.edu/~dpetrou/freebsd_lottery.tar.gz. Included are two new kernel source files, a context diff for applying patches to 14 existing kernel files, and the source for 10 userlevel programs that interact with the scheduler.

## 10 Acknowledgments

We thank UC Berkeley's Computer Science Undergraduate Association and FreeBSD Users' Group for permitting us to deploy our experimental kernel on their production machines. Peter Mardahl kindly offered `xoopic` for some of our benchmarks. We thank Aaron Smith for convincing us that emulating `nice` semantics was necessary. Thanks also go to David Greenman for providing us with statistics on `wcarchive`.

## References

[4.4BSD 1994] The 4.4BSD source code, 1994. See ftp://ftp.cdrom.com/pub/bsd-sources.

[Birrell 1989] Birrell, A. D. An introduction to programming with threads. Technical Report 35, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, January 1989.

[FreeBSD 1997] The FreeBSD Operating System, 1997. See http://www.freebsd.org/.

[Goldberg et al. 1996] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[Hauser et al. 1993] Hauser, C., Jacobi, C., Theimer, M., Welch, B., and Weiser, M. Using threads in interactive systems: A case study. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 94–105, December 5–8 1993.

[Hellerstein 1993] Hellerstein, J. L. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[Int 1996] Intel. *Pentium Pro Family Developer's Manual,* volume 2, 1996.

[Jolitz & Jolitz 1996] Jolitz, W. F. and Jolitz, L. G. *Source Code Secrets: The Basic Kernel*, volume 1. Peer-to-Peer Communications, Inc., 1996.

[Lampson & Redell 1980] Lampson, B. W. and Redell, D. D. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[Lampson 1984] Lampson, B. W. Hints for Computer System Design. *IEEE Software*, 1(1):11–28, January 1984.

[Lehey 1996] Lehey, G. *The Complete FreeBSD*. Walnut Creek, September 1996.

[Massalin & Pu 1989] Massalin, H. and Pu, C. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, volume 23, pp. 191–201, December 1989.

[McKusick et al. 1996] McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley Publishing Company, Inc., 1996.

[Nieh et al. 1994] Nieh, J., Hanko, J. G., Northcutt, J. D., and Wall, G. A. SVR4 UNIX scheduler unacceptable for multimedia applications. *Lecture Notes in Computer Science*, 846, 1994.

[Parnas 1972] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Patterson et al. 1995] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 79–95, December 3–6 1995.

[rc564 1997] Project RC5, 1997. See http://www.distributed.net/rc5/.

[Ritchie & Thompson 1974] Ritchie, D. M. and Thompson, K. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[Shneiderman 1992] Shneiderman, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley Publishing Co., Reading, MA, second edition, 1992.

[Vahalia 1996] Vahalia, U. *UNIX Internals: The New Frontiers.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[Waldspurger & Weihl 1994] Waldspurger, C. A. and Weihl, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Mangement. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pp. 1–11, November 14–17 1994.

[Waldspurger & Weihl 1996] Waldspurger, C. A. and Weihl, W. E. An object-oriented framework for modular resource management. In *Fifth Workshop on Object-Orientation in Operating Systems (IWOOOS '96), October, 1996. Seattle, WA*, 1996.

[Waldspurger 1995] Waldspurger, C. A. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD dissertation, Massachusetts Institute of Technology, September 1995.

[xoopic 1997] XOOPIC Plasma Simulation Program, 1997. See http://ptsg.eecs.berkeley.edu/xoopic/xoopic.html.