Appears in *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994

Lottery Scheduling: Flexible Proportional-Share Resource Management

Carl A. Waldspurger *   William E. Weihl *

MIT Laboratory for Computer Science
Cambridge, MA 02139 USA

Abstract

This paper presents *lottery scheduling*, a novel randomized resource allocation mechanism. Lottery scheduling provides efficient, responsive control over the relative execution rates of computations. Such control is beyond the capabilities of conventional schedulers, and is desirable in systems that service requests of varying importance, such as databases, media-based applications, and networks. Lottery scheduling also supports modular resource management by enabling concurrent modules to insulate their resource allocation policies from one another. A *currency* abstraction is introduced to flexibly name, share, and protect resource rights. We also show that lottery scheduling can be generalized to manage many diverse resources, such as I/O bandwidth, memory, and access to locks. We have implemented a prototype lottery scheduler for the Mach 3.0 microkernel, and found that it provides flexible and responsive control over the relative execution rates of a wide range of applications. The overhead imposed by our unoptimized prototype is comparable to that of the standard Mach timesharing policy.

1 Introduction

Scheduling computations in multithreaded systems is a complex, challenging problem. Scarce resources must be multiplexed to service requests of varying importance, and the policy chosen to manage this multiplexing can have an enormous impact on throughput and response time. Accurate control over the quality of service provided to users and applications requires support for specifying relative computation rates. Such control is desirable across a wide spectrum of systems. For long-running computations such as scientific applications and simulations, the consumption of computing resources that are shared among users and applications of varying importance must be regulated [Hel93]. For interactive computations such as databases and media-based applications, programmers and users need the ability to rapidly focus available resources on tasks that are currently important [Dui90].

Few general-purpose schemes even come close to supporting flexible, responsive control over service rates. Those that do exist generally rely upon a simple notion of *priority* that does not provide the encapsulation and modularity properties required for the engineering of large software systems. In fact, with the exception of hard real-time systems, it has been observed that the assignment of priorities and dynamic priority adjustment schemes are often ad-hoc [Dei90]. Even popular priority-based schemes for CPU allocation such as *decay-usage scheduling* are poorly understood, despite the fact that they are employed by numerous operating systems, including Unix [Hel93].

Existing *fair share* schedulers [Hen84, Kay88] and *microeconomic* schedulers [Fer88, Wal92] successfully address some of the problems with absolute priority schemes. However, the assumptions and overheads associated with these systems limit them to relatively coarse control over long-running computations. Interactive systems require rapid, dynamic control over scheduling at a time scale of milliseconds to seconds.

We have developed *lottery scheduling*, a novel randomized mechanism that provides responsive control over the relative execution rates of computations. Lottery scheduling efficiently implements *proportional-share* resource management — the resource consumption rates of active computations are proportional to the relative shares that they are allocated. Lottery scheduling also provides excellent support for modular resource management. We have developed a prototype lottery scheduler for the Mach 3.0 microkernel, and found that it provides efficient, flexible control over the relative execution rates of compute-bound tasks, video-based applications, and client-server interactions. This level of control is not possible with current operating systems, in which adjusting scheduling parameters to achieve specific results is at best a black art.

Lottery scheduling can be generalized to manage many diverse resources, such as I/O bandwidth, memory, and access to locks. We have developed a prototype lottery-scheduled mutex implementation, and found that it provides flexible control over mutex acquisition rates. A variant of lottery scheduling can also be used to efficiently manage space-shared resources such as memory.

*E-mail: {carl, weihl}@lcs.mit.edu. World Wide Web: http://www.psg.lcs.mit.edu/. The first author was supported in part by an AT&T USL Fellowship and by a grant from the MIT X Consortium. Prof. Weihl is currently supported by DEC while on sabbatical at DEC SRC. This research was also supported by ARPA under contract N00014-94-1-0985, by grants from AT&T and IBM, and by an equipment grant from DEC. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

In the next section, we describe the basic lottery scheduling mechanism. Section 3 discusses techniques for modular resource management based on lottery scheduling. Implementation issues and a description of our prototype are presented in Section 4. Section 5 discusses the results of several quantitative experiments. Generalizations of the lottery scheduling approach are explored in Section 6. In Section 7, we examine related work. Finally, we summarize our conclusions in Section 8.

2 Lottery Scheduling

Lottery scheduling is a randomized resource allocation mechanism. Resource rights are represented by lottery *tickets*.[1] Each allocation is determined by holding a *lottery*; the resource is granted to the client with the winning ticket. This effectively allocates resources to competing clients in proportion to the number of tickets that they hold.

2.1 Resource Rights

Lottery tickets encapsulate resource rights that are abstract, relative, and uniform. They are *abstract* because they quantify resource rights independently of machine details. Lottery tickets are *relative*, since the fraction of a resource that they represent varies dynamically in proportion to the contention for that resource. Thus, a client will obtain more of a lightly contended resource than one that is highly contended; in the worst case, it will receive a share proportional to its share of tickets in the system. Finally, tickets are *uniform* because rights for heterogeneous resources can be homogeneously represented as tickets. These properties of lottery tickets are similar to those of money in computational economies [Wal92].

2.2 Lotteries

Scheduling by lottery is *probabilistically fair*. The expected allocation of resources to clients is proportional to the number of tickets that they hold. Since the scheduling algorithm is randomized, the actual allocated proportions are not guaranteed to match the expected proportions exactly. However, the disparity between them decreases as the number of allocations increases.

The number of lotteries won by a client has a binomial distribution. The probability p that a client holding t tickets will win a given lottery with a total of T tickets is simply $p = t/T$. After n identical lotteries, the expected number of wins w is $E[w] = np$, with variance $\sigma_w^2 = np(1-p)$. The coefficient of variation for the observed proportion of wins is $\sigma_w/E[w] = \sqrt{(1-p)/np}$. Thus, a client's throughput is proportional to its ticket allocation, with accuracy that improves with $\sqrt{n}$.

[1] A single physical ticket may represent any number of logical tickets. This is similar to monetary notes, which may be issued in different denominations.

The number of lotteries required for a client's first win has a geometric distribution. The expected number of lotteries n that a client must wait before its first win is $E[n] = 1/p$, with variance $\sigma_n^2 = (1-p)/p^2$. Thus, a client's average response time is inversely proportional to its ticket allocation. The properties of both binomial and geometric distributions are well-understood [Tri82].

With a scheduling quantum of 10 milliseconds (100 lotteries per second), reasonable fairness can be achieved over subsecond time intervals. As computation speeds continue to increase, shorter time quanta can be used to further improve accuracy while maintaining a fixed proportion of scheduler overhead.

Since any client with a non-zero number of tickets will eventually win a lottery, the conventional problem of starvation does not exist. The lottery mechanism also operates fairly when the number of clients or tickets varies dynamically. For each allocation, every client is given a fair chance of winning proportional to its share of the total number of tickets. Since any changes to relative ticket allocations are immediately reflected in the next allocation decision, lottery scheduling is extremely responsive.

3 Modular Resource Management

The explicit representation of resource rights as lottery tickets provides a convenient substrate for modular resource management. Tickets can be used to insulate the resource management policies of independent modules, because each ticket probabilistically guarantees its owner the right to a worst-case resource consumption rate. Since lottery tickets abstractly encapsulate resource rights, they can also be treated as first-class objects that may be transferred in messages.

This section presents basic techniques for implementing resource management policies with lottery tickets. Detailed examples are presented in Section 5.

3.1 Ticket Transfers

Ticket transfers are explicit transfers of tickets from one client to another. Ticket transfers can be used in any situation where a client blocks due to some dependency. For example, when a client needs to block pending a reply from an RPC, it can temporarily transfer its tickets to the server on which it is waiting. This idea also conveniently solves the conventional priority inversion problem in a manner similar to priority inheritance [Sha90]. Clients also have the ability to divide ticket transfers across multiple servers on which they may be waiting.

3.2 Ticket Inflation

Ticket inflation is an alternative to explicit ticket transfers in which a client can escalate its resource rights by creating more lottery tickets. In general, such inflation should be disallowed, since it violates desirable modularity and load insulation properties. For example, a single client could easily monopolize a resource by creating a large number of lottery tickets. However, ticket inflation can be very useful among mutually trusting clients; inflation and deflation can be used to adjust resource allocations without explicit communication.

3.3 Ticket Currencies

In general, resource management abstraction barriers are desirable across logical trust boundaries. Lottery scheduling can easily be extended to express resource rights in units that are local to each group of mutually trusting clients. A unique *currency* is used to denominate tickets within each trust boundary. Each currency is backed, or *funded*, by tickets that are denominated in more primitive currencies. Currency relationships may form an arbitrary acyclic graph, such as a hierarchy of currencies. The effects of inflation can be locally contained by maintaining an *exchange rate* between each local currency and a *base* currency that is conserved. The currency abstraction is useful for flexibly naming, sharing, and protecting resource rights. For example, an access control list associated with a currency could specify which principals have permission to inflate it by creating new tickets.

3.4 Compensation Tickets

A client which consumes only a fraction f of its allocated resource quantum can be granted a *compensation ticket* that inflates its value by $1/f$ until the client starts its next quantum. This ensures that each client's resource consumption, equal to f times its per-lottery win probability p, is adjusted by $1/f$ to match its allocated share p. Without compensation tickets, a client that does not consume its entire allocated quantum would receive less than its entitled share of the processor.

4 Implementation

We have implemented a prototype lottery scheduler by modifying the Mach 3.0 microkernel (MK82) [Acc86, Loe92] on a 25MHz MIPS-based DECStation 5000/125. Full support is provided for ticket transfers, ticket inflation, ticket currencies, and compensation tickets.[2] The scheduling quantum on this platform is 100 milliseconds.

4.1 Random Numbers

An efficient lottery scheduler requires a fast way to generate uniformly-distributed random numbers. We have implemented a pseudo-random number generator based on the Park-Miller algorithm [Par88, Car90] that executes in approximately 10 RISC instructions. Our assembly-language implementation is listed in Appendix A.

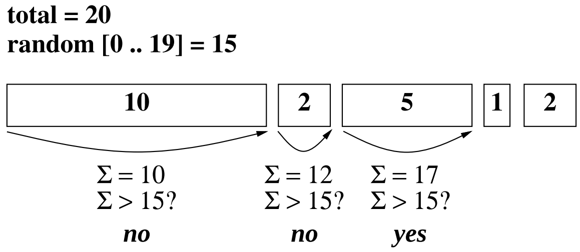

Figure 1: **Example Lottery.** Five clients compete in a list-based lottery with a total of 20 tickets. The fifteenth ticket is randomly selected, and the client list is searched for the winner. A running ticket sum is accumulated until the winning ticket value is reached. In this example, the third client is the winner.

4.2 Lotteries

A straightforward way to implement a centralized lottery scheduler is to randomly select a winning ticket, and then search a list of clients to locate the client holding that ticket. This requires a random number generation and $O(n)$ operations to traverse a client list of length n, accumulating a running ticket sum until it reaches the winning value. An example list-based lottery is presented in Figure 1.

Various optimizations can reduce the average number of clients that must be examined. For example, if the distribution of tickets to clients is uneven, ordering the clients by decreasing ticket counts can substantially reduce the average search length. Since those clients with the largest number of tickets will be selected most frequently, a simple "move to front" heuristic can be very effective.

For large n, a more efficient implementation is to use a tree of partial ticket sums, with clients at the leaves. To locate the client holding a winning ticket, the tree is traversed starting at the root node, and ending with the winning client leaf node, requiring only $O(\lg n)$ operations. Such a tree-based implementation can also be used as the basis of a distributed lottery scheduler.

4.3 Mach Kernel Interface

The kernel representation of tickets and currencies is depicted in Figure 2. A minimal lottery scheduling interface is exported by the microkernel. It consists of operations to create and destroy tickets and currencies, operations to fund and unfund a currency (by adding or removing a ticket from its list of backing tickets), and operations to compute the current value of tickets and currencies in base units.

Our lottery scheduling policy co-exists with the standard timesharing and fixed-priority policies. A few high-priority threads (such as the Ethernet driver) created by the Unix server (UX41) remain at their original fixed priorities.

[2] Our first lottery scheduler implementation, developed for the *Prelude* [Wei91] runtime system, lacked support for ticket transfers and currencies.

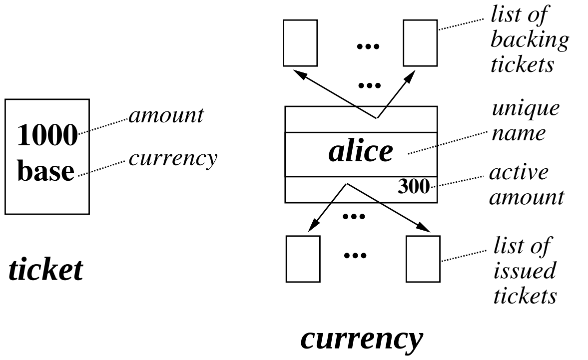

Figure 2: **Kernel Objects.** A *ticket* object contains an amount denominated in some currency. A *currency* object contains a name, a list of tickets that back the currency, a list of all tickets issued in the currency, and an active amount sum for all issued tickets.

4.4 Ticket Currencies

Our prototype uses a simple scheme to convert ticket amounts into base units. Each currency maintains an active amount sum for all of its issued tickets. A ticket is *active* while it is being used by a thread to compete in a lottery. When a thread is removed from the run queue, its tickets are deactivated; they are reactivated when the thread rejoins the run queue.[3] If a ticket deactivation changes a currency's active amount to zero, the deactivation propagates to each of its backing tickets. Similarly, if a ticket activation changes a currency's active amount from zero, the activation propagates to each of its backing tickets.

A currency's value is computed by summing the value of its backing tickets. A ticket's value is computed by multiplying the value of the currency in which it is denominated by its share of the active amount issued in that currency. The value of a ticket denominated in the base currency is defined to be its face value amount. An example currency graph with base value conversions is presented in Figure 3. Currency conversions can be accelerated by caching values or exchange rates, although this is not implemented in our prototype.

Our scheduler uses the simple list-based lottery with a move-to-front heuristic, as described earlier in Section 4.2. To handle multiple currencies, a winning ticket value is selected by generating a random number between zero and the total number of active tickets in the *base* currency. The run queue is then traversed as described earlier, except that the running ticket sum accumulates the value of each thread's currency in *base* units until the winning value is reached.

[3] A blocked thread may transfer its tickets to another thread that will actively use them. For example, a thread blocked pending a reply from an RPC transfers its tickets to the server thread on which it is waiting.

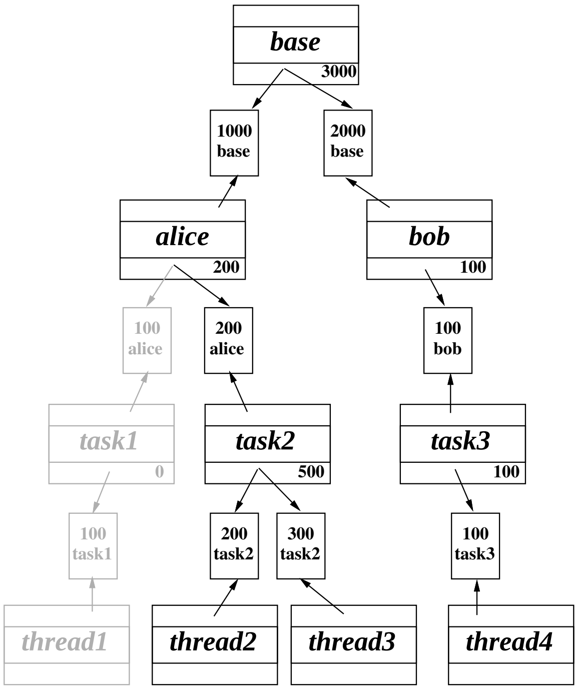

Figure 3: **Example Currency Graph.** Two users compete for computing resources. Alice is executing two tasks: *task1* is currently inactive, and *task2* has two runnable threads. Bob is executing one single-threaded task, *task3*. The current values in base units for the runnable threads are *thread2* = 400, *thread3* = 600, and *thread4* = 2000. In general, currencies can also be used for groups of users or applications, and currency relationships may form an acyclic graph instead of a strict hierarchy.

4.5 Compensation Tickets

As discussed in Section 3.4, a thread which consumes only a fraction f of its allocated time quantum is automatically granted a compensation ticket that inflates its value by $1/f$ until the thread starts its next quantum. This is consistent with proportional sharing, and permits I/O-bound tasks that use few processor cycles to start quickly.

For example, suppose threads A and B each hold tickets valued at 400 base units. Thread A always consumes its entire 100 millisecond time quantum, while thread B uses only 20 milliseconds before yielding the processor. Since both A and B have equal funding, they are equally likely to win a lottery when both compete for the processor. However, thread B uses only $f = 1/5$ of its allocated time, allowing thread A to consume five times as much CPU, in violation of their $1:1$ allocation ratio. To remedy this situation, thread B is granted a compensation ticket valued at 1600 base units when it yields the processor. When B next competes for the processor, its total funding will be $400/f = 2000$ base units. Thus, on average B will win the processor lottery five times as often as A, each time consuming $1/5$ as much of its quantum as A, achieving the desired $1:1$ allocation ratio.

4.6 Ticket Transfers

The `mach_msg` system call was modified to temporarily transfer tickets from client to server for synchronous RPCs. This automatically redirects resource rights from a blocked client to the server computing on its behalf. A transfer is implemented by creating a new ticket denominated in the client's currency, and using it to fund the server's currency. If the server thread is already waiting when `mach_msg` performs a synchronous call, it is immediately funded with the transfer ticket. If no server thread is waiting, then the transfer ticket is placed on a list that is checked by the server thread when it attempts to receive the call message.[4] During a reply, the transfer ticket is simply destroyed.

4.7 User Interface

Currencies and tickets can be manipulated via a command-line interface. User-level commands exist to create and destroy tickets and currencies (`mktkt`, `rmtkt`, `mkcur`, `rmcur`), fund and unfund currencies (`fund`, `unfund`), obtain information (`lstkt`, `lscur`), and to execute a shell command with specified funding (`fundx`). Since the Mach microkernel has no concept of user and we did not modify the Unix server, these commands are setuid root.[5] A complete lottery scheduling system should protect currencies by using access control lists or Unix-style permissions based on user and group membership.

5 Experiments

In order to evaluate our prototype lottery scheduler, we conducted experiments designed to quantify its ability to flexibly, responsively, and efficiently control the relative execution rates of computations. The applications used in our experiments include the compute-bound Dhrystone benchmark, a Monte-Carlo numerical integration program, a multithreaded client-server application for searching text, and competing MPEG video viewers.

5.1 Fairness

Our first experiment measured the accuracy with which our lottery scheduler could control the relative execution rates of computations. Each point plotted in Figure 4 indicates the relative execution rate that was observed for two tasks executing the Dhrystone benchmark [Wei84] for sixty seconds with a given relative ticket allocation. Three runs were executed for each integral ratio between one and ten.

[4] In this case, it would be preferable to instead fund all threads capable of receiving the message. For example, a server task with fewer threads than incoming messages should be directly funded. This would accelerate all server threads, decreasing the delay until one becomes available to service the waiting message.

[5] The `fundx` command only executes as root to initialize its task currency funding. It then performs a setuid back to the original user before invoking `exec`.

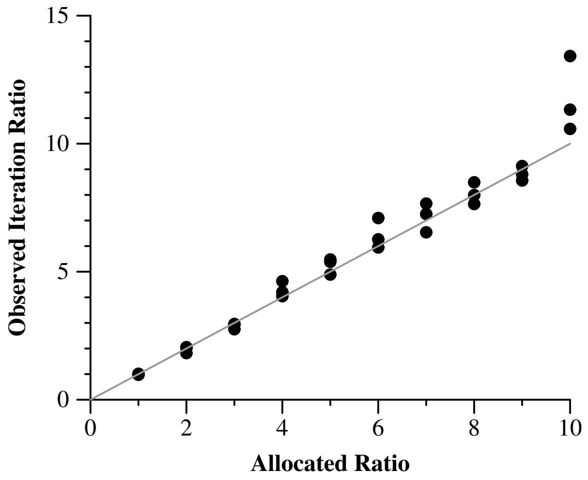

Figure 4: **Relative Rate Accuracy.** For each allocated ratio, the observed ratio is plotted for each of three 60 second runs. The gray line indicates the ideal where the two ratios are identical.

With the exception of the run for which the 10 : 1 allocation resulted in an average ratio of 13.42 : 1, all of the observed ratios are close to their corresponding allocations. As expected, the variance is greater for larger ratios. However, even large ratios converge toward their allocated values over longer time intervals. For example, the observed ratio averaged over a three minute period for a 20 : 1 allocation was 19.08 : 1.

Although the results presented in Figure 4 indicate that the scheduler can successfully control computation rates, we should also examine its behavior over shorter time intervals. Figure 5 plots average iteration counts over a series of 8 second time windows during a single 200 second execution with a 2 : 1 allocation. Although there is clearly some variation, the two tasks remain close to their allocated ratios throughout the experiment. Note that if a scheduling quantum of 10 milliseconds were used instead of the 100 millisecond Mach quantum, the same degree of fairness would be observed over a series of subsecond time windows.

5.2 Flexible Control

A more interesting use of lottery scheduling involves dynamically controlled ticket inflation. A practical application that benefits from such control is the Monte-Carlo algorithm [Pre88]. Monte-Carlo is a probabilistic algorithm that is widely used in the physical sciences for computing average properties of systems. Since errors in the computed average are proportional to $1/\sqrt{n}$, where n is the number of trials, accurate results require a large number of trials.

Scientists frequently execute several separate Monte-Carlo experiments to explore various hypotheses. It is often desirable to obtain approximate results quickly whenever a new experiment is started, while allowing older experiments to continue reducing their error at a slower rate [Hog88].

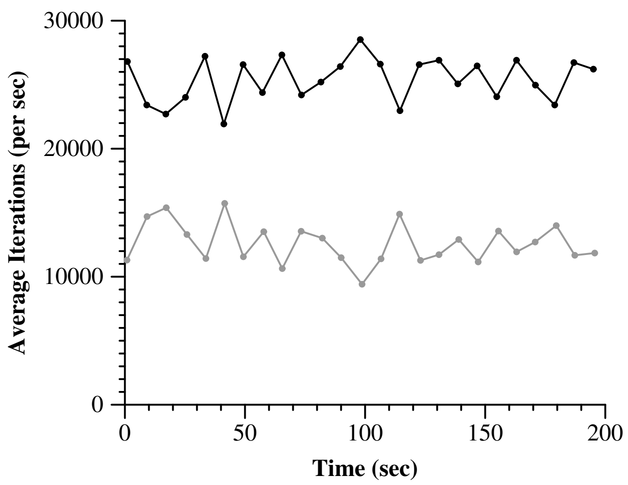

Figure 5: **Fairness Over Time.** Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations/sec., for an actual ratio of 2.01 : 1.

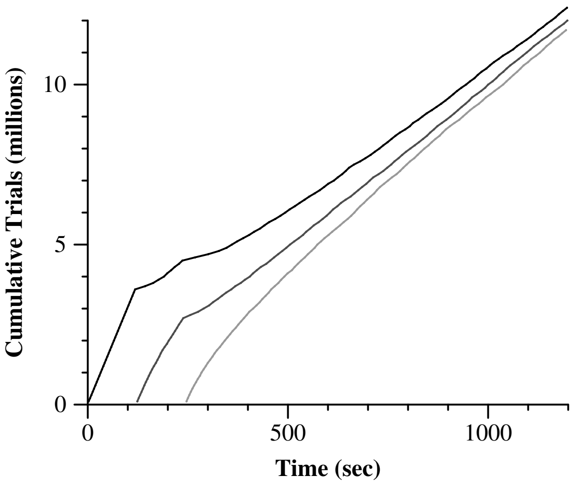

Figure 6: **Monte-Carlo Execution Rates.** Three identical Monte-Carlo integrations are started two minutes apart. Each task periodically sets its ticket value to be proportional to the square of its relative error, resulting in the convergent behavior. The "bumps" in the curves mirror the decreasing slopes of new tasks that quickly reduce their error.

This goal would be impossible with conventional schedulers, but can be easily achieved in our system by dynamically adjusting an experiment's ticket value as a function of its current relative error. This allows a new experiment with high error to quickly catch up to older experiments by executing at a rate that starts high but then tapers off as its relative error approaches that of its older counterparts.

Figure 6 plots the total number of trials computed by each of three staggered Monte-Carlo tasks. Each task is based on the sample code presented in [Pre88], and is allocated a share of time that is proportional to the square of its relative error.[6] When a new task is started, it initially receives a large share of the processor. This share diminishes as the task reduces its error to a value closer to that of the other executing tasks.

A similar form of dynamic control may also be useful in graphics-intensive programs. For example, a rendering operation could be granted a large share of processing resources until it has displayed a crude outline or wire-frame, and then given a smaller share of resources to compute a more polished image.

5.3 Client-Server Computation

As mentioned in Section 4.6, the Mach IPC primitive `mach_msg` was modified to temporarily transfer tickets from client to server on synchronous remote procedure calls. Thus, a client automatically redirects its resource rights to the server that is computing on its behalf. Multithreaded servers will process requests from different clients at the rates defined by their respective ticket allocations.

We developed a simple multithreaded client-server application that shares properties with real databases and information retrieval systems. Our server initially loads a 4.6 Mbyte text file "database" containing the complete text to all of William Shakespeare's plays.[7] It then forks off several worker threads to process incoming queries from clients. One query operation supported by the server is a case-insensitive substring search over the entire database, which returns a count of the matches found.

Figure 7 presents the results of executing three database clients with an 8 : 3 : 1 ticket allocation. The server has no tickets of its own, and relies completely upon the tickets transferred by clients. Each client repeatedly sends requests to the server to count the occurrences of the same search string.[8] The high-priority client issues a total of 20 queries and then terminates. The other two clients continue to issue queries for the duration of the entire experiment.

The ticket allocations affect both response time and throughput. When the high-priority client has completed its 20 requests, the other clients have completed a total of 10 requests, matching their overall 8 : 4 allocation. Over the entire experiment, the clients with a 3 : 1 ticket allocation respectively complete 38 and 13 queries, which closely matches their allocation, despite their transient competition with the high-priority client. While the high-priority client is active, the average response times seen by the clients are 17.19, 43.19, and 132.20 seconds, yielding relative speeds of 7.69 : 2.51 : 1. After the high-priority client terminates,

[6] Any monotonically increasing function of the relative error would cause convergence. A linear function would cause the tasks to converge more slowly; a cubic function would result in more rapid convergence.

[7] A disk-based database could use lotteries to schedule disk bandwidth; this is not implemented in our prototype.

[8] The string used for this experiment was `lottery`, which incidentally occurs a total of 8 times in Shakespeare's plays.

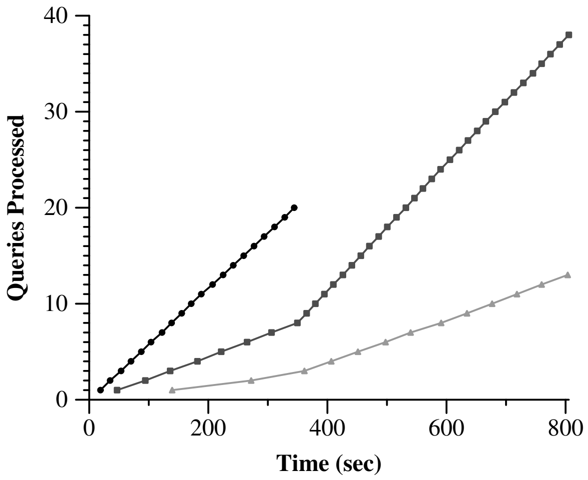

Figure 7: **Query Processing Rates.** Three clients with an 8 : 3 : 1 ticket allocation compete for service from a multithreaded database server. The observed throughput and response time ratios closely match this allocation.

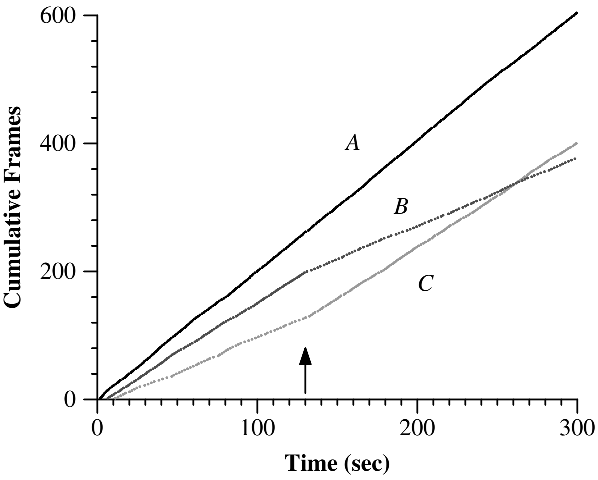

Figure 8: **Controlling Video Rates.** Three MPEG viewers are given an initial $A:B:C = 3:2:1$ allocation, which is changed to 3 : 1 : 2 at the time indicated by the arrow. The total number of frames displayed is plotted for each viewer. The actual frame rate ratios were 1.92 : 1.50 : 1 and 1.92 : 1 : 1.53, respectively, due to distortions caused by the X server.

the response times are 44.17 and 15.18 seconds, for a 2.91 : 1 ratio. For all average response times, the standard deviation is less than 7% of the average.

A similar form of control could be employed by database or transaction-processing applications to manage the response times seen by competing clients or transactions. This would be useful in providing different levels of service to clients or transactions with varying importance (or real monetary funding).

5.4 Multimedia Applications

Media-based applications are another domain that can benefit from lottery scheduling. Compton and Tennenhouse described the need to control the quality of service when two or more video viewers are displayed — a level of control not offered by current operating systems [Com94]. They attempted, with mixed success, to control video display rates at the application level among a group of mutually trusting viewers. Cooperating viewers employed feedback mechanisms to adjust their relative frame rates. Inadequate and unstable metrics for system load necessitated substantial tuning, based in part on the number of active viewers. Unexpected positive feedback loops also developed, leading to significant divergence from intended allocations.

Lottery scheduling enables the desired control at the operating-system level, eliminating the need for mutually trusting or well-behaved applications. Figure 8 depicts the execution of three `mpeg_play` video viewers (A, B, and C) displaying the same music video. Tickets were initially allocated to achieve relative display rates of $A:B:C = 3:2:1$, and were then changed to 3 : 1 : 2 at the time indicated by the arrow. The observed per-second frame rates were initially 2.03 : 1.59 : 1.06 (1.92 : 1.50 : 1 ratio), and then 2.02 : 1.05 : 1.61 (1.92 : 1 : 1.53 ratio) after the change. Unfortunately, these results were distorted by the round-robin processing of client requests by the single-threaded X11R5 server. When run with the `-no_display` option, frame rates such as 6.83 : 4.56 : 2.23 (3.06 : 2.04 : 1 ratio) were typical.

5.5 Load Insulation

Support for multiple ticket currencies facilitates modular resource management. A currency defines a resource management abstraction barrier that locally contains intra-currency fluctuations such as inflation. The currency abstraction can be used to flexibly isolate or group users, tasks, and threads.

Figure 9 plots the progress of five tasks executing the Dhrystone benchmark. Let *amount.currency* denote a ticket allocation of *amount* denominated in *currency*. Currencies A and B have identical funding. Tasks $A1$ and $A2$ have allocations of $100.A$ and $200.A$, respectively. Tasks $B1$ and $B2$ have allocations of $100.B$ and $200.B$, respectively. Halfway through the experiment, a new task, $B3$, is started with an allocation of $300.B$. Although this inflates the total number of tickets denominated in currency B from 300 to 600, there is no effect on tasks in currency A. The aggregate iteration ratio of A tasks to B tasks is 1.01 : 1 before $B3$ is started, and 1.00 : 1 after $B3$ is started. The slopes for the individual tasks indicate that $A1$ and $A2$ are not affected by task $B3$, while $B1$ and $B2$ are slowed to approximately half their original rates, corresponding to the factor of two inflation caused by $B3$.

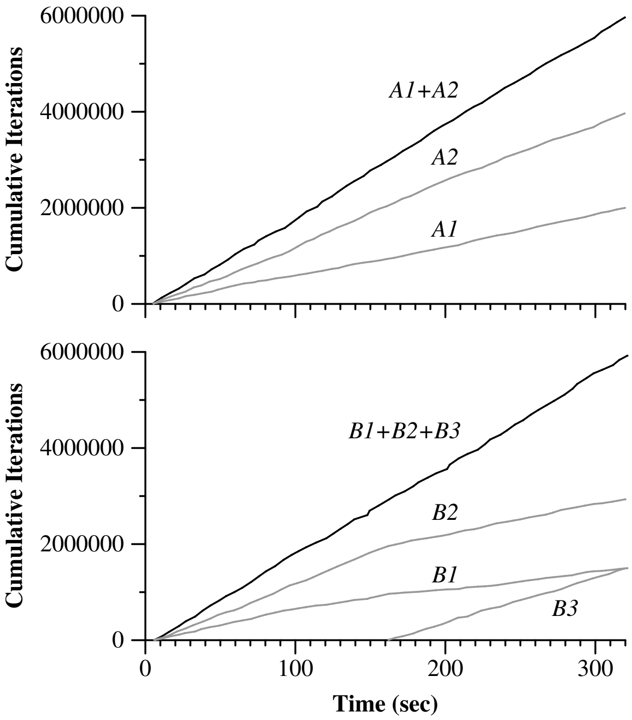

Figure 9: **Currencies Insulate Loads.** Currencies A and B are identically funded. Tasks $A1$ and $A2$ are respectively allocated tickets worth $100.A$ and $200.A$. Tasks $B1$ and $B2$ are respectively allocated tickets worth $100.B$ and $200.B$. Halfway through the experiment, task $B3$ is started with an allocation of $300.B$. The resulting inflation is locally contained within currency B, and affects neither the progress of tasks in currency A, nor the aggregate $A:B$ progress ratio.

5.6 System Overhead

The core lottery scheduling mechanism is extremely lightweight; a tree-based lottery need only generate a random number and perform $\lg n$ additions and comparisons to select a winner among n clients. Thus, low-overhead lottery scheduling is possible in systems with a scheduling granularity as small as a thousand RISC instructions.

Our prototype scheduler, which includes full support for currencies, has not been optimized. To assess system overhead, we used the same executables and workloads under both our kernel and the unmodified Mach kernel; three separate runs were performed for each experiment. Overall, we found that the overhead imposed by our prototype lottery scheduler is comparable to that of the standard Mach timesharing policy. Since numerous optimizations could be made to our list-based lottery, simple currency conversion scheme, and other untuned aspects of our implementation, efficient lottery scheduling does not pose any challenging problems.

Our first experiment consisted of three Dhrystone benchmark tasks running concurrently for 200 seconds. Compared to unmodified Mach, 2.7% fewer iterations were executed under lottery scheduling. For the same experiment with eight tasks, lottery scheduling was observed to be 0.8% slower. However, the standard deviations across individual runs for unmodified Mach were comparable to the absolute differences observed between the kernels. Thus, the measured differences are not very significant.

We also ran a performance test using the multithreaded database server described in Section 5.3. Five client tasks each performed 20 queries, and the time between the start of the first query and the completion of the last query was measured. We found that this application executed 1.7% faster under lottery scheduling. For unmodified Mach, the average run time was 1155.5 seconds; with lottery scheduling, the average time was 1135.5 seconds. The standard deviations across runs for this experiment were less than 0.1% of the averages, indicating that the small measured differences are significant.[9]

6 Managing Diverse Resources

Lotteries can be used to manage many diverse resources, such as processor time, I/O bandwidth, and access to locks. Lottery scheduling also appears promising for scheduling communication resources, such as access to network ports. For example, ATM switches schedule virtual circuits to determine which buffered cell should next be forwarded. Lottery scheduling could be used to provide different levels of service to virtual circuits competing for congested channels. In general, a lottery can be used to allocate resources wherever queueing is necessary for resource access.

6.1 Synchronization Resources

Contention due to synchronization can substantially affect computation rates. Lottery scheduling can be used to control the relative waiting times of threads competing for lock access. We have extended the Mach CThreads library to support a lottery-scheduled mutex type in addition to the standard mutex implementation. A lottery-scheduled mutex has an associated *mutex currency* and an *inheritance ticket* issued in that currency.

All threads that are blocked waiting to acquire the mutex perform ticket transfers to fund the mutex currency. The mutex transfers its inheritance ticket to the thread which currently holds the mutex. The net effect of these transfers is that a thread which acquires the mutex executes with its own funding plus the funding of all waiting threads, as depicted in Figure 10. This solves the priority inversion problem [Sha90], in which a mutex owner with little funding could execute very slowly due to competition with other threads

[9] Under unmodified Mach, threads with equal priority are run round-robin; with lottery scheduling, it is possible for a thread to win several lotteries in a row. We believe that this ordering difference may affect locality, resulting in slightly improved cache and TLB behavior for this application under lottery scheduling.

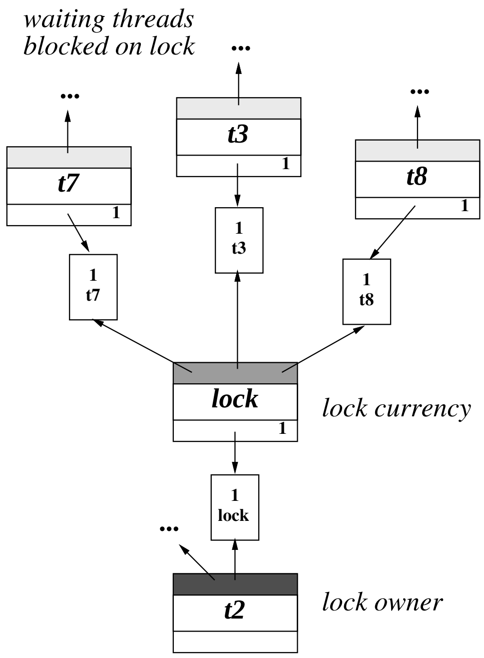

Figure 10: **Lock Funding.** Threads *t3*, *t7*, and *t8* are waiting to acquire a lottery-scheduled lock, and have transferred their funding to the lock currency. Thread *t2* currently holds the lock, and inherits the aggregate waiter funding through the backing ticket denominated in the lock currency. Instead of showing the backing tickets associated with each thread, shading is used to indicate relative funding levels.

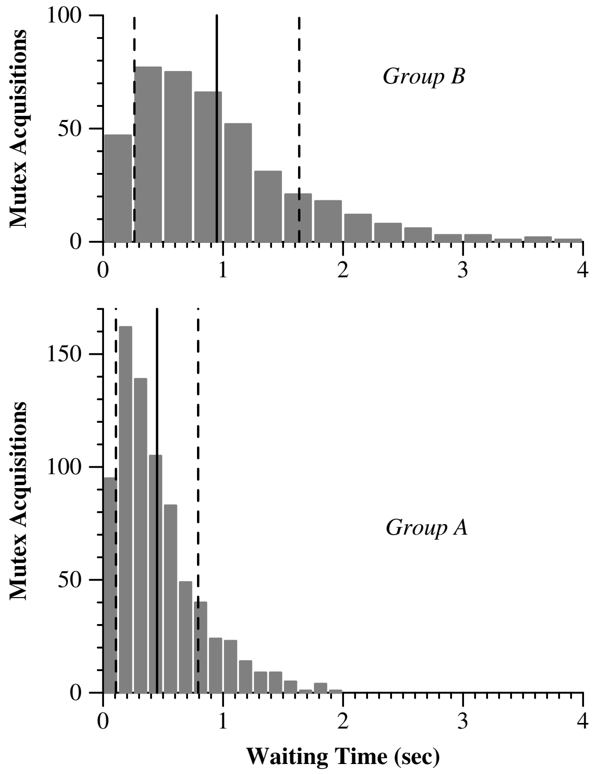

Figure 11: **Mutex Waiting Times.** Eight threads compete to acquire a lottery-scheduled mutex. The threads are divided into two groups (A, B) of four threads each, with the ticket allocation $A : B = 2 : 1$. For each histogram, the solid line indicates the mean (μ); the dashed lines indicate one standard deviation about the mean ($\mu \pm \sigma$). The ratio of average waiting times is $A : B = 1 : 2.11$; the mutex acquisition ratio is $1.80 : 1$.

for the processor, while a highly funded thread remains blocked on the mutex.

When a thread releases a lottery-scheduled mutex, it holds a lottery among the waiting threads to determine the next mutex owner. The thread then moves the mutex inheritance ticket to the winner, and yields the processor. The next thread to execute may be the selected waiter or some other thread that does not need the mutex; the normal processor lottery will choose fairly based on relative funding.

We have experimented with our mutex implementation using a synthetic multithreaded application in which n threads compete for the same mutex. Each thread repeatedly acquires the mutex, holds it for h milliseconds, releases the mutex, and computes for another t milliseconds. Figure 11 provides frequency histograms for a typical experiment with $n = 8$, $h = 50$, and $t = 50$. The eight threads were divided into two groups (A, B) of four threads each, with the ticket allocation $A : B = 2 : 1$. Over the entire two-minute experiment, group A threads acquired the mutex a total of 763 times, while group B threads completed 423 acquisitions, for a relative throughput ratio of $1.80 : 1$. The group A threads had a mean waiting time of $\mu = 450$ milliseconds , while the group B threads had a mean waiting time of $\mu = 948$ milliseconds, for a relative waiting time ratio of $1 : 2.11$. Thus, both throughput and response time closely tracked the specified $2 : 1$ ticket allocation.

6.2 Space-Shared Resources

Lotteries are useful for allocating indivisible time-shared resources, such as an entire processor. A variant of lottery scheduling can efficiently provide the same type of probabilistic proportional-share guarantees for finely divisible space-shared resources, such as memory. The basic idea is to use an *inverse lottery*, in which a "loser" is chosen to relinquish a unit of a resource that it holds. Conducting an inverse lottery is similar to holding a normal lottery, except that inverse probabilities are used. The probability p that a client holding t tickets will be selected by an inverse lottery with a total of n clients and T tickets is $p = \frac{1}{n-1}(1 - t/T)$. Thus, the more tickets a client has, the more likely it is to avoid having a unit of its resource revoked.[10]

For example, consider the problem of allocating a physical page to service a virtual memory page fault when all

[10] The $\frac{1}{n-1}$ factor is a normalization term which ensures that the client probabilities sum to unity.

physical pages are in use. A proportional-share policy based on inverse lotteries could choose a client from which to select a victim page with probability proportional to both $(1 - t/T)$ and the fraction of physical memory in use by that client.

6.3 Multiple Resources

Since rights for numerous resources are uniformly represented by lottery tickets, clients can use quantitative comparisons to make decisions involving tradeoffs between different resources. This raises some interesting questions regarding application funding policies in environments with multiple resources. For example, when does it make sense to shift funding from one resource to another? How frequently should funding allocations be reconsidered?

One way to abstract the evaluation of resource management options is to associate a separate *manager* thread with each application. A manager thread could be allocated a small fixed percentage (*e.g.*, 1%) of an application's overall funding, causing it to be periodically scheduled while limiting its overall resource consumption. For inverse lotteries, it may be appropriate to allow the losing client to execute a short manager code fragment in order to adjust funding levels. The system should supply default managers for most applications; sophisticated applications could define their own management strategies. We plan to explore these preliminary ideas and other alternatives for more complex environments with multiple resources.

7 Related Work

Conventional operating systems commonly employ a simple notion of *priority* in scheduling tasks. A task with higher priority is given absolute precedence over a task with lower priority. Priorities may be static, or they may be allowed to vary dynamically. Many sophisticated priority schemes are somewhat arbitrary, since priorities themselves are rarely meaningfully assigned [Dei90]. The ability to express priorities provides absolute, but extremely crude, control over scheduling, since resource rights do not vary smoothly with priorities. Conventional priority mechanisms are also inadequate for insulating the resource allocation policies of separate modules. Since priorities are absolute, it is difficult to compose or abstract inter-module priority relationships.

Fair share schedulers allocate resources so that users get fair machine shares over long periods of time [Hen84, Kay88]. These schedulers monitor CPU usage and dynamically adjust conventional priorities to push actual usage closer to entitled shares. However, the algorithms used by these systems are complex, requiring periodic usage updates, complicated dynamic priority adjustments, and administrative parameter setting to ensure fairness on a time scale of minutes. A technique also exists for achieving service rate objectives in systems that employ *decay-usage scheduling* by manipulating base priorities and various scheduler parameters [Hel93]. While this technique avoids the addition of feedback loops introduced by other fair share schedulers, it still assumes a fixed workload consisting of long-running compute-bound processes to ensure steady-state fairness at a time scale of minutes.

Microeconomic schedulers [Dre88, Fer88, Wal92] use auctions to allocate resources among clients that bid monetary funds. Funds encapsulate resource rights and serve as a form of priority. Both the *escalator algorithm* proposed for uniprocessor scheduling [Dre88] and the distributed *Spawn* system [Wal89, Wal92] rely upon auctions in which bidders increase their bids linearly over time. The *Spawn* system successfully allocated resources proportional to client funding in a network of heterogeneous workstations. However, experience with *Spawn* revealed that auction dynamics can be unexpectedly volatile. The overhead of bidding also limits the applicability of auctions to relatively coarse-grain tasks.

A market-based approach for memory allocation has also been developed to allow memory-intensive applications to optimize their memory consumption in a decentralized manner [Har92]. This scheme charges applications for both memory *leases* and I/O capacity, allowing application-specific tradeoffs to be made. However, unlike a true market, prices are not permitted to vary with demand, and ancillary parameters are introduced to restrict resource consumption [Che93].

The *statistical matching* technique for fair switching in the AN2 network exploits randomness to support frequent changes of bandwidth allocation [And93]. This work is similar to our proposed application of lottery scheduling to communication channels.

8 Conclusions

We have presented lottery scheduling, a novel mechanism that provides efficient and responsive control over the relative execution rates of computations. Lottery scheduling also facilitates modular resource management, and can be generalized to manage diverse resources. Since lottery scheduling is conceptually simple and easily implemented, it can be added to existing operating systems to provide greatly improved control over resource consumption rates. We are currently exploring various applications of lottery scheduling in interactive systems, including graphical user interface elements. We are also examining the use of lotteries for managing memory, virtual circuit bandwidth, and multiple resources.

Acknowledgements

We would like to thank Kavita Bala, Eric Brewer, Dawson Engler, Wilson Hsieh, Bob Gruber, Anthony Joseph, Frans Kaashoek, Ulana Legedza, Paige Parsons, Patrick

Sobalvarro, and Debby Wallach for their comments and assistance. Special thanks to Kavita for her invaluable help with Mach, and to Anthony for his patient critiques of several drafts. Thanks also to Jim Lipkis and the anonymous reviewers for their many helpful suggestions.

References

[Acc86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Conference*, June 1986.

[And93] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. "High-Speed Switch Scheduling for Local-Area Networks," *ACM Transactions on Computer Systems*, November 1993.

[Car90] D. G. Carta. "Two Fast Implementations of the 'Minimal Standard' Random Number Generator," *Communications of the ACM*, January 1990.

[Che93] D. R. Cheriton and K. Harty. "A Market Approach to Operating System Memory Allocation," Working Paper, Computer Science Department, Stanford University, June 1993.

[Com94] C. L. Compton and D. L. Tennenhouse. "Collaborative Load Shedding for Media-based Applications," *Proceedings of the International Conference on Multimedia Computing and Systems*, May 1994.

[Dei90] H. M. Deitel. *Operating Systems*, Addison-Wesley, 1990.

[Dre88] K. E. Drexler and M. S. Miller. "Incentive Engineering for Computational Resource Management" in *The Ecology of Computation*, B. Huberman (ed.), North-Holland, 1988.

[Dui90] D. Duis and J. Johnson. "Improving User-Interface Responsiveness Despite Performance Limitations," *Proceedings of the Thirty-Fifth IEEE Computer Society International Conference (COMPCON)*, March 1990.

[Fer88] D. Ferguson, Y. Yemini, and C. Nikolaou. "Microeconomic Algorithms for Load-Balancing in Distributed Computer Systems," *International Conference on Distributed Computer Systems*, 1988.

[Har92] K. Harty and D. R. Cheriton. "Application-Controlled Physical Memory using External Page-Cache Management," *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Hel93] J. L. Hellerstein. "Achieving Service Rate Objectives with Decay Usage Scheduling," *IEEE Transactions on Software Engineering*, August 1993.

[Hen84] G. J. Henry. "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal*, October 1984.

[Hog88] T. Hogg. Private communication (during *Spawn* system development), 1988.

[Kan89] G. Kane. *Mips RISC Architecture*, Prentice-Hall, 1989.

[Kay88] J. Kay and P. Lauder. "A Fair Share Scheduler," *Communications of the ACM*, January 1988.

[Loe92] K. Loepere. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University, 1992.

[Par88] S. K. Park and K. W. Miller. "Random Number Generators: Good Ones Are Hard to Find," *Communications of the ACM*, October 1988.

[Pre88] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1988.

[Sha90] L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, September 1990.

[Tri82] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, 1982.

[Wal89] C. A. Waldspurger. "A Distributed Computational Economy for Utilizing Idle Resources," Master's thesis, MIT, May 1989.

[Wal92] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. "Spawn: A Distributed Computational Economy," *IEEE Transactions on Software Engineering*, February 1992.

[Wei84] R. P. Weicker. "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, October 1984.

[Wei91] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. "Prelude: A System for Portable Parallel Software," Technical Report MIT/LCS/TR-519, MIT Lab for Computer Science, October 1991.

A Random Number Generator

This MIPS assembly-language code [Kan89] is a fast implementation of the Park-Miller pseudo-random number generator [Par88, Car90]. It uses the multiplicative linear congruential generator $S' = (A \times S) \bmod (2^{31} - 1)$, for $A = 16807$. The generator's ANSI C prototype is: `unsigned int fastrand(unsigned int s)`.

```
fastrand:
    move    $2, $4          | R2 = S (arg passed in R4)
    li      $8, 33614       | R8 = 2 × constant A
    multu   $2, $8          | HI, LO = A × S
    mflo    $9              | R9  = Q  = bits 00..31 of A × S
    srl     $9, $9, 1
    mfhi    $10             | R10 = P  = bits 32..63 of A × S
    addu    $2, $9, $10     | R2  = S' = P + Q
    bltz    $2, overflow    | handle overflow (rare)
    j       $31             | return (result in R2)

overflow:
    sll     $2, $2, 1       | zero bit 31 of S'
    srl     $2, $2, 1
    addiu   $2, 1           | increment S'
    j       $31             | return (result in R2)
```