Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management

by

Carl A. Waldspurger

S.B., Massachusetts Institute of Technology (1989)
S.M., Massachusetts Institute of Technology (1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1995

Author . . . .
Department of Electrical Engineering and Computer Science
September 5, 1995

Certified by . . . .
William E. Weihl
Associate Professor
Thesis Supervisor

Accepted by . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Lottery and Stride Scheduling:

Flexible Proportional-Share Resource Management

by

Carl A. Waldspurger

Submitted to the Department of Electrical Engineering and Computer Science
on September 5, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis presents flexible abstractions for specifying resource management policies, together with efficient mechanisms for implementing those abstractions. Several novel scheduling techniques are introduced, including both randomized and deterministic algorithms that provide proportional-share control over resource consumption rates. Such control is beyond the capabilities of conventional schedulers, and is desirable across a broad spectrum of systems that service clients of varying importance. Proportional-share scheduling is examined for several diverse resources, including processor time, memory, access to locks, and disk bandwidth.

Resource rights are encapsulated by abstract, first-class objects called *tickets*. An active client consumes resources at a rate proportional to the number of tickets that it holds. Tickets can be issued in different amounts and may be transferred between clients. A modular *currency* abstraction is also introduced to flexibly name, share, and protect sets of tickets. Currencies can be used to isolate or group sets of clients, enabling the modular composition of arbitrary resource management policies.

Two different underlying mechanisms are introduced to support these abstractions. *Lottery scheduling* is a novel randomized resource allocation mechanism. An allocation is performed by holding a *lottery*, and the resource is granted to the client with the winning ticket. *Stride scheduling* is a deterministic resource allocation mechanism that computes a representation of the time interval, or *stride*, that each client must wait between successive allocations. Stride scheduling cross-applies and generalizes elements of rate-based flow control algorithms designed for networks to dynamically schedule other resources such as processor time. A novel *hierarchical* stride algorithm is also introduced that achieves better throughput accuracy than prior schemes, and can reduce response-time variability for some workloads.

The proposed techniques are compared and evaluated using a variety of quantitative experiments. Simulation results and prototype implementations for operating system kernels demonstrate flexible control over a wide range of resources and applications.

Thesis Supervisor: William E. Weihl
Title: Associate Professor

Acknowledgments

Thanks to Bill Weihl, my thesis advisor, for providing the right mix of guidance and freedom to help me grow as a computer scientist. His insights and suggestions have contributed greatly to this dissertation, and I look forward to continued collaboration in the future. Bill is a great advisor and a terrific person.

Thanks also to my other thesis readers, Frans Kaashoek and Bill Dally, for their critical feedback and advice that helped to improve the overall quality of this dissertation.

Special thanks to Paige Parsons for her help with all aspects of this thesis. Paige acted as a sounding board for many of my early ideas and provided invaluable assistance with both textual and graphical presentations. I can't adequately express in words my appreciation for all of Paige's encouragement, love, and support. Our three house rabbit companions Foobar, Zowie, and Zippy also deserve thanks for daily doses of playfulness and affection.

Thanks to Pat Parsons for carefully proofreading this thesis and catching more errors than I care to admit. She also provided some much-appreciated levity by interspersing humor with corrections in numerous e-mail messages.

I would also like to express my gratitude to all of the friends and officemates with whom I've spent many years as a graduate student at MIT. Thanks to Kavita Bala, Eric Brewer, Dawson Engler, Sanjay Ghemawat, Bob Gruber, Wilson Hsieh, Anthony Joseph, Bruce Leban, Ulana Legedza, Patrick Sobalvarro, and Debby Wallach. Our frequent discussions of everything from concurrency to liberty made my time at MIT more enjoyable. Anthony alone is probably responsible for several months of thesis avoidance by provoking arguments with questions like "As a vegetarian, would you eat a sea cucumber? Why not?". Special thanks to Kavita for her invaluable help with Mach, and to Anthony for his assistance with Linux.

Thanks to Bernardo Huberman and the members of the Dynamics of Computation group at Xerox PARC who shared their enthusiasm for research with me as I worked on my master's thesis. They were also the first to focus my attention on interesting problems related to computational resource management.

Since the completion of this dissertation marks the end of one score and three years as a student, I would also like to thank everyone who contributed to my pre-MIT education. Past teachers who stand out as significant influences include Joseph Oechsle, Jim Roche, and Ernie McCabe. In addition, I am grateful to the members of the LaSalle Forum who helped sharpen my logic and presentation abilities.

Given my early experiences with computers, it is fitting that randomized algorithms are included in my dissertation. I would like to thank Frank Grazian for sparking an early interest in probability and computation; one of the first real programs that I wrote simulated his strategies for baccarat. I also credit Don Bagin with fostering an interest in computers and gaming by exposing me to the *Daily Racing Form*. As a high school sophomore, I developed a program to automate his sprint handicapping system.

In retrospect, my advocacy of proportional-share control shouldn't be too surprising either. Even as a young child I remember my father wondering why people continued to design confusing nonlinearities into everything from radio-controlled toys to gas gauges in cars. I feel the same way about scheduling parameters for computer systems.

I am deeply grateful to my father for imparting his appreciation and understanding of science and technology. I am also indebted to my mother for her tireless efforts to ensure that I received the best possible education. Both my parents made countless sacrifices to finance my education. We were also among the first on our block to get a home computer, when most folks opted instead for newer cars or furniture. This was during the pre-IBM-PC era when one of my high school teachers actually needed to be convinced that my home computer was only *formatting* my reports, and not *writing* them. Overall, I am extremely fortunate to be part of such a close and wonderful family. None of this would have been possible without the love, humor, and support of my parents and my brothers, Roy and Scott.

To Paige

Contents

List of Figures

Chapter 1

Introduction

Scheduling computations in concurrent systems is a complex, challenging problem. 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. Effective resource management requires knowledge of both user preferences and application-specific performance characteristics. Unfortunately, users, applications, and other clients of resources are typically given very limited control over resource management policies. Traditional operating systems centrally manage machine resources within the kernel [EKO95]. Clients are commonly afforded only crude control through poorly understood, ad-hoc scheduling parameters. Worse yet, such parameters do not offer the encapsulation or modularity properties required for the engineering of large software systems.

This thesis advocates a radically different approach to computational resource management. Resource rights are treated as first-class objects representing well-defined resource shares. Clients are permitted to directly redistribute their resource rights in order to control computation rates. In addition, a simple, powerful abstraction is provided to facilitate modular composition of resource management policies. As a result, custom policies can be expressed conveniently at various levels of abstraction. The role of the operating system in resource management is reduced to one of enforcement, ensuring that no client is able to consume more than its entitled share of resources.

This chapter presents a high-level synopsis of the thesis. The next section contains a basic overview of the key thesis components. This is followed by highlights of the main contributions of the thesis, and a brief summary of related research. The chapter closes with a description of the overall organization for the rest of the thesis.

1.1 Overview

Accurate control over service rates is desirable across a broad spectrum of systems that process requests of varying importance. For long-running computations such as scientific applications and simulations, the consumption of computing resources that are shared among different users and applications must be regulated. 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.

This thesis proposes a general framework for specifying dynamic resource management policies, together with efficient mechanisms for implementing that framework. Resource rights are encapsulated by abstract, first-class objects called *tickets*. An active client is entitled to consume resources at a rate proportional to the number of tickets that it holds. Tickets can be issued in different amounts and may be transferred between clients. A modular *currency* abstraction is also introduced to flexibly name, share, and protect sets of tickets. Currencies can be used to isolate or group sets of clients, enabling the modular composition of arbitrary resource management policies.

Two different underlying proportional-share mechanisms are introduced to support this framework. *Lottery scheduling* is a novel randomized resource allocation mechanism. An allocation is performed by holding a *lottery*, and the resource is granted to the client with the winning ticket. *Stride scheduling* is a deterministic resource allocation mechanism that computes a representation of the time interval, or *stride*, that each client must wait between successive allocations. Stride scheduling cross-applies and generalizes elements of rate-based flow control algorithms designed for networks [DKS90, Zha91, ZK91, PG93] to dynamically schedule other resources such as processor time. Novel variants of these core mechanisms are also introduced to provide improved proportional-share accuracy for many workloads. In addition, a number of new resource-specific techniques are proposed for proportional-share management of diverse resources including memory, access to locks, and disk bandwidth.

The proposed proportional-share techniques are compared and evaluated using a variety of quantitative experiments. Extensive simulation results and prototype process-scheduler implementations for real operating system kernels demonstrate flexible control over a wide range of resources and applications. The overall system overhead imposed by these unoptimized prototypes is comparable to that of the default timesharing policies that they replaced.

1.2 Contributions

This thesis makes several research contributions: a versatile new framework for specifying resource management policies, novel algorithms for proportional-share control over time-shared resources, and specialized techniques for managing other resource classes. The general resource management framework is based on direct, proportional-share control over service rates using tickets and currencies. Its principal features include:

- *Simplicity*: An intuitive notion of relative resource shares is used instead of complex, non-linear, or ad-hoc scheduling parameters. Resource rights vary smoothly with ticket allocations, allowing precise control over computation rates. The resource rights represented by tickets also aggregate in a natural additive manner.

- *Modularity*: Modularity is key to good software engineering practices. Currencies provide explicit support for modular abstraction of resource rights. The currency abstraction is analogous to class-based abstraction of data in object-oriented languages with multiple inheritance. Collections of tickets can be named, shared, and protected in a modular way. This enables the resource management policies of concurrent modules to be insulated from one another, facilitating modular decomposition.

- *Flexibility*: Sophisticated patterns of sharing and protection can be conveniently expressed for resource rights, including hierarchical organizations and relationships defined by more general acyclic graphs. Resource management policies can be defined for clients at various levels of abstraction, such as threads, applications, users, and groups.

- *Adaptability*: Client service rates degrade gracefully in overload situations, and active clients benefit proportionally from extra resources when some allocations are not fully utilized. These properties facilitate adaptive applications that can respond to changes in resource availability.

- *Generality*: The framework is intended for general-purpose computer systems, and is not dependent on restrictive assumptions about clients. The same general framework can be applied to a wide range of diverse resources. It can also serve as a solid foundation for simultaneously managing multiple heterogeneous resources.

An implementation of this general framework requires proportional-share scheduling algorithms that efficiently support dynamic environments. Another contribution of this thesis is the development of several new algorithms for proportional-share scheduling of time-shared resources. Both randomized and deterministic mechanisms are introduced:

- *Lottery scheduling*: A novel randomized resource allocation mechanism that inherently supports dynamic environments. Lottery scheduling is conceptually simple and easily implemented. However, it exhibits poor throughput accuracy over short allocation intervals, and produces high response-time variability for low-throughput clients.

- *Multi-winner lottery scheduling*: A variant of lottery scheduling that produces better throughput accuracy and lower response-time variability for many workloads.

- *Stride scheduling*: A deterministic resource allocation mechanism that implements dynamic, proportional-share control over processor time and other resources. Compared to the randomized lottery-based approaches, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly lower response-time variability.

- *Hierarchical stride scheduling*: A novel recursive application of the basic stride scheduling technique that provides a tighter bound on throughput accuracy than ordinary stride scheduling. Hierarchical stride scheduling can also reduce response-time variability for some workloads.

Additional contributions include new resource-specific techniques for dynamic, proportional-share scheduling of diverse resources. The following mechanisms were developed to manage synchronization resources, space-shared resources, and disk I/O bandwidth:

- *Ticket inheritance*: An extension to the basic algorithms for time-shared resources to schedule synchronization resources such as lock accesses. This technique enables proportional-share control over computation rates despite contention for locks.

- *Inverse lottery scheduling*: A variant of lottery scheduling for dynamic, revocation-based management of space-shared resources such as memory. A randomized inverse lottery selects a “loser” that is forced to relinquish a resource unit.

- *Minimum-funding revocation*: A simple deterministic scheme for proportional-share control over space-shared resources. A resource unit is revoked from the client expending the fewest tickets per resource unit. Compared to randomized inverse lottery scheduling, minimum funding revocation is more efficient and converges toward proportional shares more rapidly.

- *Funded delay cost scheduling*: A deterministic disk scheduling algorithm presented as a first step toward proportional-share control over disk bandwidth.

1.3 Related Work

This section places the research described in this thesis in context by presenting an overview of related work. Computational resource management techniques from a variety of fields are briefly summarized; a more complete discussion appears in Chapter 7.

The dominant processor scheduling paradigm in operating systems is *priority scheduling* [Dei90, Tan92]. Conventional timesharing policies employ dynamic priority adjustment schemes based on ad-hoc, non-linear functions that are poorly understood. Manipulating scheduling parameters to achieve specific results in such systems is at best a black art.[1] Attempts to control service rates using timesharing schedulers have been largely unsuccessful, providing only coarse, limited control [Hel93, FS95]. Priority schedulers also lack desirable modularity properties that are essential for good software engineering practices.

Fair share schedulers attempt to allocate resources so that users get fair machine shares over long periods of time [Hen84, KL88, Hel93]. These schedulers are layered on top of conventional priority schedulers, and dynamically adjust priorities to push actual usage closer to entitled shares. The algorithms used by these systems are generally complex, requiring periodic usage monitoring, complicated dynamic priority adjustments, and administrative parameter tuning to ensure fairness on a time scale of minutes.

Despite their ad-hoc treatment in most operating systems, priorities are used in a clear and consistent manner in the domain of *real-time systems* [BW90]. Real-time schedulers must ensure that absolute scheduling deadlines are met in order to ensure correctness and avoid catastrophic failures [Bur91]. The widely-used *rate-monotonic scheduling* technique [LL73, SKG91] statically assigns priorities as a monotonic function of the rate of periodic tasks. A task's priority does not depend on its importance; tasks with shorter periods are always assigned higher priorities. Another common technique is *earliest deadline scheduling* [LL73], which always schedules the task with the closest deadline first. Higher-level abstractions based on real-time scheduling have also been developed [MST93, MST94]. However, real-time schedulers generally depend upon very restrictive assumptions, such as precise static knowledge of task execution times and prohibitions on task interactions. Strict limits are also placed on processor utilization, so that even transient overloads are disallowed.

A number of deterministic *proportional-share scheduling* algorithms have recently been proposed [BGP95, FS95, Mah95, SAW95]. Several of these techniques [FS95, Mah95, SAW95] make explicit comparisons to lottery scheduling [WW94], although none of them demonstrate support for the higher-level abstractions introduced with lottery scheduling. In general, the

[1] Anyone who has had the misfortune of trying to implement precise scheduling behavior by setting Unix *nice* values can attest to this fact.

proportional-share accuracy of these schedulers is better than lottery scheduling, and comparable to stride scheduling. However, the algorithms used by these schedulers require expensive operations to transform client state in response to dynamic changes. Since dynamic operations cannot be implemented efficiently, these approaches are not suitable for supporting the general resource management framework proposed in this thesis.

Proportional-share algorithms have also been designed for *rate-based flow control* in packet-switched networks [DKS90, Zha91, ZK91, PG93]. The core stride scheduling algorithm presented in this thesis essentially cross-applies and extends elements of the *virtual clock* [Zha91] and *weighted fair queueing* [DKS90] algorithms to the domain of dynamic processor scheduling. To the best of my knowledge, stride scheduling is the first cross-application of these techniques for scheduling resources other than network bandwidth. The hierarchical stride scheduling algorithm introduced in this thesis is a novel recursive application of the stride-based technique, extended for dynamic environments. An unrelated scheme from the domain of network traffic management is also similar to randomized lottery scheduling. The *statistical matching* technique proposed for the AN2 network exploits randomness to efficiently support frequent dynamic changes in bandwidth allocations [AOST93].

Microeconomic schedulers are based on resource allocation techniques in real economic systems [MD88, HH95a, Wel95]. *Money* encapsulates resource rights, and a *price* mechanism is used to allocate resources. Microeconomic schedulers [DM88, Fer89, WHH$^+$92, Wel93, Bog94] typically use auctions to determine prices and allocate resources among clients that bid monetary funds. However, auction dynamics can be unexpectedly volatile, and bidding overhead limits the applicability of auctions to relatively coarse-grained tasks. Other market-based approaches that do not rely upon auctions have also been applied to managing processor and memory resources [Ell75, HC92, CH93]. The framework and mechanisms proposed in this thesis are compatible with a market-based resource management philosophy.

1.4 Organization

This section previews the remaining chapters, and describes the overall organization of the thesis. The next chapter presents a general framework for specifying resource management policies in concurrent systems. The use of tickets and currencies is shown to facilitate flexible, modular control over resource management. Chapter 3 introduces several scheduling algorithms that can be used as a substrate for implementing this framework. Both randomized lottery-based techniques and deterministic stride-based approaches are presented for achieving proportional-share control over resource consumption rates. Chapter 4 examines and compares the performance of these scheduling techniques in both static and dynamic environments.

Performance is evaluated by deriving basic analytical results and by conducting a wide range of quantitative simulation experiments.

Prototype implementations of proportional-share process schedulers for real operating system kernels are described in Chapter 5. The results of quantitative experiments involving a variety of user-level applications are presented to demonstrate flexible, responsive control over application service rates. Chapter 6 considers the application of proportional-share scheduling techniques to diverse resources, including memory, disk bandwidth, and access to locks. Extensions to the core scheduling techniques are presented, and several novel resource-specific algorithms are also introduced. Chapter 7 discusses a wide variety of research related to computational resource management in much greater detail than the brief summary presented in this chapter. Finally, Chapter 8 summarizes the conclusions of this thesis and highlights opportunities for additional research.

Chapter 2

Resource Management Framework

This chapter presents a general, flexible framework for specifying resource management policies in concurrent systems. Resource rights are encapsulated by abstract, first-class objects called *tickets*. Ticket-based policies are expressed using two basic techniques: *ticket transfers* and *ticket inflation*. Ticket transfers allow resource rights to be directly transferred and redistributed among clients. Ticket inflation allows resource rights to be changed by manipulating the overall supply of tickets. A powerful *currency* abstraction provides flexible, modular control over ticket inflation. Currencies also support the sharing, protecting, and naming of resource rights. Several example resource management policies are presented to demonstrate the versatility of this framework.

2.1 Tickets

Resource rights are encapsulated by first-class objects called *tickets*. Tickets can be issued in different amounts, so that a single physical ticket may represent any number of logical tickets. In this respect, tickets are similar to monetary notes which are also issued in different denominations. For example, a single ticket object may represent one hundred tickets, just as a single $100 bill represents one hundred separate $1 bills.

Tickets are owned by *clients* that consume resources. A client is considered to be *active* while it is competing to acquire more resources. An active client is entitled to consume resources at a rate proportional to the number of tickets that it has been allocated. Thus, a client with twice as many tickets as another is entitled to receive twice as much of a resource in a given time interval. The number of tickets allocated to a client also determines its entitled response time. Client response times are defined to be inversely proportional to ticket allocations. Therefore, a client with twice as many tickets as another is entitled to wait only half as long before acquiring a resource.

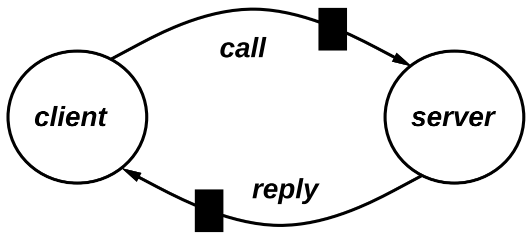

Figure 2-1: **Example Ticket Transfer.** A client performs a ticket transfer to a server during a synchronous remote procedure call (RPC). The server executes with the resource rights of the client, and then returns those rights during the RPC reply.

Tickets encapsulate resource rights that are abstract, relative, and uniform. Tickets are *abstract* because they quantify resource rights independently of machine details. 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, a client will receive a share proportional to its share of tickets in the system. This property facilitates adaptive clients that can benefit from extra resources when other clients do not fully utilize their allocations. Finally, tickets are *uniform* because rights for heterogeneous resources can be homogeneously represented as tickets. This property permits clients to use quantitative comparisons when making decisions that involve tradeoffs between different resources.

In general, tickets have properties that are similar to those of money in computational economies [WHH$^+$92]. The only significant difference is that tickets are not consumed when they are used to acquire resources. A client may reuse a ticket any number of times, but a ticket may only be used to compete for one resource at a time. In economic terms, a ticket behaves much like a constant monetary income stream.

2.2 Ticket Transfers

A *ticket transfer* is an explicit transfer of first-class ticket objects from one client to another. Ticket transfers can be used to implement resource management policies by directly redistributing resource rights. Transfers are useful in any situation where one client blocks waiting for another. For example, Figure 2-1 illustrates the use of a ticket transfer during a synchronous remote procedure call (RPC). A client performs a temporary ticket transfer to loan its resource rights to the server computing on its behalf.

Ticket transfers also provide a convenient solution to the conventional priority inversion problem in a manner that is similar to priority inheritance [SRL90]. For example, clients waiting to acquire a lock can temporarily transfer tickets to the current lock owner. This provides the lock owner with additional resource rights, helping it to obtain a larger share of processor time so that it can more quickly release the lock. Unlike priority inheritance, transfers from multiple clients are additive. A client also has the flexibility to split ticket transfers across multiple clients on which it may be waiting. These features would not make sense in a priority-based system, since resource rights do not vary smoothly with priorities.

Ticket transfers are capable of specifying *any* ticket-based resource management policy, since transfers can be used to implement any arbitrary distribution of tickets to clients. However, ticket transfers are often too low-level to conveniently express policies. The exclusive use of ticket transfers imposes a *conservation* constraint: tickets may be redistributed, but they cannot be created or destroyed. This constraint ensures that no client can deprive another of resources without its permission. However, it also complicates the specification of many natural policies.

For example, consider a set of processes, each a client of a time-shared processor resource. Suppose that a parent process spawns child subprocesses and wants to allocate resource rights equally to each child. To achieve this goal, the parent must explicitly coordinate ticket transfers among its children whenever a child process is created or destroyed. Although ticket transfers alone are capable of supporting arbitrary resource management policies, their specification is often unnecessarily complex.

2.3 Ticket Inflation and Deflation

Ticket inflation and *deflation* are alternatives to explicit ticket transfers. Client resource rights can be escalated by creating more tickets, inflating the total number of tickets in the system. Similarly, client resource rights can be reduced by destroying tickets, deflating the overall number of tickets. Ticket inflation and deflation are useful among mutually trusting clients, since they permit resource rights to be reallocated without explicitly reshuffling tickets among clients. This can greatly simplify the specification of many resource management policies. For example, a parent process can allocate resource rights equally to child subprocesses simply by creating and assigning a fixed number of tickets to each child that is spawned, and destroying the tickets owned by each child when it terminates.

However, uncontrolled ticket inflation is dangerous, since a client can monopolize a resource by creating a large number of tickets. Viewed from an economic perspective, inflation is a form of theft, since it devalues the tickets owned by all clients. Because inflation can violate desirable modularity and insulation properties, it must be either prohibited or strictly controlled.

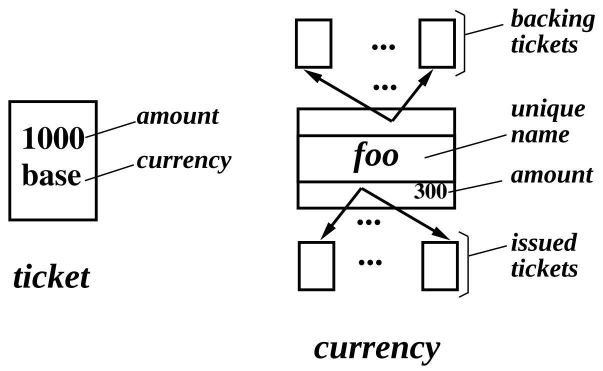

Figure 2-2: **Ticket and Currency Objects.** A *ticket* object contains an amount denominated in some currency. A *currency* object contains a name, a list of backing tickets that *fund* the currency, a list of all tickets issued in the currency, and an amount that contains the total number of active tickets issued in the currency.

A key observation is that the desirability of inflation and deflation hinges on *trust*. Trust implies permission to appropriate resources without explicit authorization. When trust is present, explicit ticket transfers are often more cumbersome and restrictive than simple, local ticket inflation. When trust is absent, misbehaving clients can use inflation to plunder resources. Distilled into a single principle, ticket inflation and deflation should be allowed only within logical *trust boundaries*. The next section introduces a powerful abstraction that can be used to define trust boundaries and safely exploit ticket inflation.

2.4 Ticket Currencies

A *ticket currency* is a resource management abstraction that contains the effects of ticket inflation in a modular way. The basic concept of a ticket is extended to include a currency in which the ticket is denominated. Since each ticket is denominated in a currency, resource rights can be expressed in units that are local to each group of mutually trusting clients. A currency derives its value from *backing tickets* that are denominated in more primitive currencies. The tickets that back a currency are said to *fund* that currency. The value of a currency can be used to fund other currencies or clients by issuing tickets denominated in that currency. The effects of inflation are locally contained by effectively maintaining an *exchange rate* between each local currency and a common *base* currency that is conserved. The values of tickets denominated in different currencies are compared by first converting them into units of the base currency.

Figure 2-2 depicts key aspects of ticket and currency objects. A ticket object consists of an amount denominated in some currency; the notation *amount.currency* will be used to refer

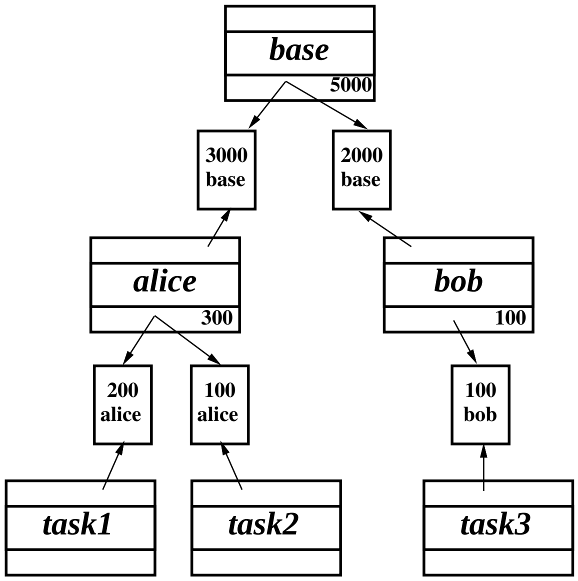

Figure 2-3: **Example Currency Graph.** Two users compete for computing resources. Alice is executing two tasks, *task1* and *task2*. Bob is executing a single task, *task3*. The current values in base units for these tasks are *task1* = 2000, *task2* = 1000, and *task3* = 2000. In general, currency relationships may form an acyclic graph instead of a strict hierarchy.

to a ticket. A currency object consists of a unique name, a list of backing tickets that fund the currency, a list of tickets issued in the currency, and an amount that contains the total number of active tickets issued in the currency. In addition, each currency should maintain permissions that determine which clients have the right to create and destroy tickets denominated in that currency. A variety of well-known schemes can be used to implement permissions [Tan92]. For example, an access control list can be associated with each currency to specify those clients that have permission to inflate it by creating new tickets.

Currency relationships may form an arbitrary acyclic graph, enabling a wide variety of different resource management policies. One useful currency configuration is a hierarchy of currencies. Each currency divides its value into subcurrencies that recursively subdivide and distribute that value by issuing tickets. Figure 2-3 presents an example currency graph with a hierarchical tree structure. In addition to the common base currency at the root of the tree, distinct currencies are associated with each user and task. Two users, Alice and Bob, are competing for computing resources. The *alice* currency is backed by 3000 tickets denominated in the base currency (3000.*base*), and the *bob* currency is backed by 2000 tickets denominated in the base currency (2000.*base*). Thus, Alice is entitled to 50% more resources than Bob, since their currencies are funded at a 3 : 2 ratio.

Alice is executing two tasks, *task1* and *task2*. She subdivides her allocation between these tasks in a 2 : 1 ratio using tickets denominated in her own currency – 200.*alice* and 100.*alice*. Since a total of 300 tickets are issued in the *alice* currency, backed by a total of 3000 base tickets, the exchange rate between the *alice* and *base* currencies is 1 : 10. Bob is executing a single task, *task3*, and uses his entire allocation to fund it via a single 100.*bob* ticket. Since a total of 100 tickets are issued in the *bob* currency, backed by a total of 2000 base tickets, the *bob* : *base* exchange rate is 1 : 20. If Bob were to create a second task with equal funding by issuing another 100.*bob* ticket, this exchange rate would become 1 : 10.

The currency abstraction is useful for flexibly sharing, protecting, and naming resource rights. Sharing is supported by allowing clients with proper permissions to inflate or deflate a currency by creating or destroying tickets. For example, a group of mutually trusting clients can form a currency that pools its collective resource rights in order to simplify resource management. Protection is guaranteed by maintaining exchange rates that automatically adjust for intra-currency fluctuations that result from internal inflation or deflation. Currencies also provide a convenient way to name resource rights at various levels of abstraction. For example, currencies can be used to name the resource rights allocated to arbitrary collections of threads, tasks, applications, or users.

Since there is nothing comparable to a currency abstraction in conventional operating systems, it is instructive to examine similar abstractions that are provided in the domain of programming languages. Various aspects of currencies can be related to features of object-oriented systems, including data abstraction, class definitions, and multiple inheritance.

For example, currency abstractions for resource rights resemble data abstractions for data objects. Data abstractions hide and protect representations by restricting access to an abstract data type. By default, access is provided only through abstract operations exported by the data type. The code that implements those abstract operations, however, is free to directly manipulate the underlying representation of the abstract data type. Thus, an *abstraction barrier* is said to exist between the abstract data type and its underlying representation [LG86]. A currency defines a resource management abstraction barrier that provides similar properties for resource rights. By default, clients are not trusted, and are restricted from interfering with resource management policies that distribute resource rights within a currency. The clients that implement a currency's resource management policy, however, are free to directly manipulate and redistribute the resource rights associated with that currency.

The use of currencies to structure resource-right relationships also resembles the use of classes to structure object relationships in object-oriented systems that support multiple inheritance. A class inherits its behavior from a set of superclasses, which are combined and modified to specify new behaviors for instances of that class. A currency inherits its funding from a set of backing tickets, which are combined and then redistributed to specify allocations for tickets denominated in that currency. However, one difference between currencies and classes is the relationship among the objects that they instantiate. When a currency issues a new ticket, it effectively dilutes the value of all existing tickets denominated in that currency. In contrast, the objects instantiated by a class need not affect one another.

2.5 Example Policies

A wide variety of resource management policies can be specified using the general framework presented in this chapter. This section examines several different resource management scenarios, and demonstrates how appropriate policies can be specified.

2.5.1 Basic Policies

Unlike priorities which specify absolute precedence constraints, tickets are specifically designed to specify relative service rates. Thus, the most basic examples of ticket-based resource management policies are simple service rate specifications. If the total number of tickets in a system is fixed, then a ticket allocation directly specifies an *absolute* share of a resource. For example, a client with 125 tickets in a system with a total of 1000 tickets will receive a 12.5% resource share. Ticket allocations can also be used to specify *relative* importance. For example, a client that is twice as important as another is simply given twice as many tickets.

Ticket inflation and deflation provide a convenient way for concurrent clients to implement resource management policies. For example, cooperative (AND-parallel) clients can independently adjust their ticket allocations based upon application-specific estimates of remaining work. Similarly, competitive (OR-parallel) clients can independently adjust their ticket allocations based on application-specific metrics for progress. One concrete example is the management of concurrent computations that perform heuristic searches. Such computations typically assign numerical values to summarize the progress made along each search path. These values can be used directly as ticket assignments, focusing resources on those paths which are most promising, without starving the exploration of alternative paths.

Tickets can also be used to fund speculative computations that have the potential to accelerate a program's execution, but are not required for correctness. With relatively small ticket allocations, speculative computations will be scheduled most frequently when there is little contention for resources. During periods of high resource contention, they will be scheduled very infrequently. Thus, very low service rate specifications can exploit unused resources while limiting the impact of speculation on more important computations.

If desired, tickets can also be used to approximate absolute priority levels. For example, a series of currencies $c_1, c_2, \ldots, c_n$ can be defined such that currency c_i has 100 times the funding of currency c_{i-1}. A client with emulated priority level i is allocated a single ticket denominated in currency c_i. Clients at priority level i will be serviced 100 times more frequently than clients at level $i - 1$, approximating a strict priority ordering.

2.5.2 Administrative Policies

For long-running computations such as those found in engineering and scientific environments, there is a need to regulate the consumption of computing resources that are shared among users and applications of varying importance [Hel93]. Currencies can be used to isolate the policies of projects, users, and applications from one another, and relative funding levels can be used to specify importance.

For example, a system administrator can allocate ticket levels to different groups based on criteria such as project importance, resource needs, or real monetary funding. Groups can subdivide their allocations among users based upon need or status within the group; an egalitarian approach would give each user an equal allocation. Users can directly allocate their own resource rights to applications based upon factors such as relative importance or impending deadlines. Since currency relationships need not follow a strict hierarchy, users may belong to multiple groups. It is also possible for one group to subsidize another. For example, if group A is waiting for results from group B, it can issue a ticket denominated in currency A, and use it to fund group B.

2.5.3 Interactive Application Policies

For interactive computations such as databases and media-based applications, programmers and users need the ability to rapidly focus resources on those tasks that are currently important. In fact, research in computer-human interaction has demonstrated that responsiveness is often the most significant factor in determining user productivity [DJ90].

Many interactive systems, such as databases and the World Wide Web, are structured using a client-server framework. Servers process requests from a wide variety of clients that may demand different levels of service. Some requests may be inherently more important or time-critical than others. Users may also vary in importance or willingness to pay a monetary premium for better service. In such scenarios, ticket allocations can be used to specify importance, and ticket transfers can be used to allow servers to compute using the resource rights of requesting clients.

Another scenario that is becoming increasingly common is the need to control the quality of service when two or more video viewers are displayed [CT94]. Adaptive viewers are capable of dynamically altering image resolution and frame rates to match current resource availability. Coupled with dynamic ticket inflation, adaptive viewers permit users to selectively improve the quality of those video streams to which they are currently paying the most attention. For example, a graphical control associated with each viewer could be manipulated to smoothly improve or degrade a viewer's quality of service by inflating or deflating its ticket allocation. Alternatively, a preset number of tickets could be associated with the window that owns the current input focus. Dynamic ticket transfers make it possible to shift resources as the focus changes, *e.g.,* in response to mouse movements. With an input device capable of tracking eye movements, a similar technique could even be used to automatically adjust the performance of applications based upon the user's visual focal point.

In addition to user-directed control over resource management, programmatic application-level control can also be used to improve responsiveness despite resource limitations [DJ90, TL93]. For example, a graphics-intensive program could devote a large share of its processing resources to a rendering operation until it has displayed a crude but usable outline or wire-frame. The share of resources devoted to rendering could then be reduced via ticket deflation, allowing a more polished image to be computed while most resources are devoted to improving the responsiveness of more critical operations.

Chapter 3

Proportional-Share Mechanisms

This chapter presents mechanisms that can be used to efficiently implement the resource management framework described in Chapter 2. Several novel scheduling algorithms are introduced, including both randomized and deterministic techniques that provide proportional-share control over time-shared resources. The algorithms are presented in the order that they were developed, followed by a discussion of their application to the general resource management framework.

One common theme is the desire to achieve proportional sharing with a high degree of accuracy. The throughput accuracy of a proportional-share scheduler can be characterized by measuring the difference between the specified and actual number of allocations that a client receives during a series of allocations. If a client has t tickets in a system with a total of T tickets, then its *specified* allocation after n_a consecutive allocations is $n_a t/T$. Due to quantization, it is typically impossible to achieve this ideal exactly. A client's *absolute error* is defined as the absolute value of the difference between its specified and actual number of allocations. The pairwise *relative error* between clients c_i and c_j is defined as the absolute error for the subsystem containing only c_i and c_j, where $T = t_i + t_j$, and n_a is the total number of allocations received by both clients.

Another key issue is the challenge of providing efficient, systematic support for dynamic operations, such as modifications to ticket allocations, and changes in the number of clients competing for a resource. Support for fast dynamic operations is also required for low-overhead implementations of higher-level abstractions such as ticket transfers, ticket inflation, and ticket currencies. Many proportional-share mechanisms that are perfectly reasonable for static environments exhibit ad-hoc behavior or unacceptable performance in dynamic environments.

After initial experimentation with a variety of different techniques, I discovered that randomization could be exploited to avoid most of the complexity associated with dynamic operations. This realization led to the development of *lottery scheduling*, a new randomized resource allocation mechanism [WW94]. Lottery scheduling performs an allocation by holding a *lottery*; the resource is granted to the client with the winning ticket. Due to its inherent use of randomization, a client's expected relative error and expected absolute error under lottery scheduling are both $O(\sqrt{n_a})$. Thus, lottery scheduling can exhibit substantial variability over small numbers of allocations. Attempts to limit this variability resulted in an investigation of *multi-winner lottery scheduling*, a hybrid technique with both randomized and deterministic components.

A desire for even more predictable behavior over shorter time scales prompted a renewed effort to develop a deterministic algorithm with efficient support for dynamic operations. Optimization of an inefficient algorithm that I originally developed before the conception of lottery scheduling resulted in *stride scheduling* [WW95]. Stride scheduling is a deterministic algorithm that computes a representation of the time interval, or *stride*, that each client must wait between successive allocations. Under stride scheduling, the relative error for any pair of clients is never greater than *one*, independent of n_a. However, for skewed ticket distributions it is still possible for a client to have $O(n_c)$ absolute error, where n_c is the number of clients.

I later discovered that the core allocation algorithm used in stride scheduling is nearly identical to elements of rate-based flow-control algorithms designed for packet-switched networks [DKS90, Zha91, ZK91, PG93]. Thus, stride scheduling can be viewed as a cross-application of these networking algorithms to schedule other resources such as processor time. However, the original network-oriented algorithms did not address the issue of dynamic operations, such as changes to ticket allocations. Since these operations are extremely important in domains such as processor scheduling, I developed new techniques to efficiently support them. These techniques can also be used to support frequent changes in bandwidth allocations for networks.

Finally, dissatisfaction with the schedules produced by stride scheduling for skewed ticket distributions led to an improved *hierarchical stride scheduling* algorithm that provides a tighter $O(\lg n_c)$ bound on each client's absolute error. Hierarchical stride scheduling is a novel recursive application of the basic technique that achieves better throughput accuracy than previous schemes, and can reduce response-time variability for some workloads.

The remainder of this chapter presents lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. Each mechanism is described in a separate section that begins with a description of the basic algorithm, followed by a discussion of extensions that support dynamic operations and irregular quantum sizes. Source code and examples are included to illustrate each mechanism. The chapter concludes by demonstrating that each presented mechanism is capable of serving as a substrate for the general resource management framework presented in Chapter 2. Detailed simulation results, performance analyses, and comparisons of the mechanisms are presented in Chapter 4.

3.1 Lottery Scheduling

Lottery scheduling is a randomized resource allocation mechanism for time-shared resources. Each allocation is determined by holding a *lottery* that randomly selects a winning ticket from the set of all tickets competing for a resource. The resource is granted to the client that holds the winning ticket. This simple operation effectively allocates resources to competing clients in proportion to the number of tickets that they hold. This section first presents the basic lottery scheduling algorithm, and then introduces extensions that support dynamic operations and nonuniform quanta.

3.1.1 Basic Algorithm

The core lottery scheduling idea is to randomly select a ticket from the set of all tickets competing for a resource. Since each ticket has an equal probability of being selected, the probability that a particular client will be selected is directly proportional to the number of tickets that it has been assigned.

In general, there are n_c clients competing for a resource, and each client c_i has t_i tickets. Thus, there are a total of $T = \sum_{i=1}^{n_c} t_i$ tickets competing for the resource. The probability p_i that client c_i will win a particular lottery is simply t_i/T. After n_a identical allocations, the expected number of wins w_i for client c_i is $E[w_i] = n_a\, p_i$, with variance $\sigma_{w_i}^2 = n_a p_i(1 - p_i)$. Thus, 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. More precisely, a client's expected relative error and expected absolute error are both $O(\sqrt{n_a})$. Since error increases slowly with n_a, accuracy steadily improves when error is measured as a percentage of n_a.

One straightforward way to implement a lottery scheduler is to randomly select a winning ticket, and then search a list of clients to locate the client holding that ticket. Figure 3-1 presents an example list-based lottery. Five clients are competing for a resource with a total of 20 tickets. The thirteenth ticket is randomly chosen, and the client list is searched to determine the client holding the winning ticket. In this example, the third client is the winner, since its region of the ticket space contains the winning ticket.

Figure 3-2 lists ANSI C code for a basic list-based lottery scheduler. For simplicity, it is assumed that the set of clients is static, and that client ticket assignments are fixed. These restrictions will be relaxed in subsequent sections to permit more dynamic behavior. Each client must be initialized via *client_init()* before any allocations are performed by *allocate()*. The *allocate()* operation begins by calling *fast_random()* to generate a uniformly-distributed

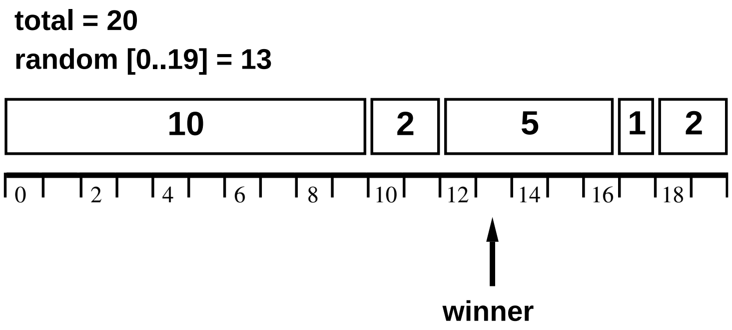

Figure 3-1: **Example List-Based Lottery.** Five clients compete in a list-based lottery with a total of 20 tickets. The thirteenth ticket is randomly selected, and the client list is searched for the winner. In this example, the third client is the winner.

```c
/* per-client state */
typedef struct {
  ...
  int tickets;
} *client_t;

/* current resource owner */
client_t current;

/* list of clients competing for resource */
list_t list;

/* global ticket sum */
int global_tickets = 0;

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* initialize client state, update global sum */
  c->tickets = tickets;
  global_tickets += tickets;

  /* join competition for resource */
  list_insert(list, c);
}

/* proportional-share resource allocation */
void allocate()
{
  int winner, sum;
  client_t c;

  /* randomly select winning ticket */
  winner = fast_random() % global_tickets;

  /* search list to find client with winning ticket */
  sum = 0;
  for (c = list_first(list);
       c != NULL;
       c = list_next(list, c))
    {
      /* update running sum, stop at winner */
      sum += c->tickets;
      if (sum > winner)
        break;
    }

  /* grant resource to winner for quantum */
  current = c;
  use_resource(current);
}
```

Figure 3-2: **List-Based Lottery Scheduling Algorithm.** ANSI C code for scheduling a static set of clients using a list-based lottery. An allocation requires $O(n_c)$ time to search the list of clients for the winning ticket. A simple doubly-linked list can be used to implement constant-time list operations.

pseudo-random integer. Numerous techniques exist for generating random numbers. For example, the Park-Miller generator efficiently produces high-quality random numbers that are uniformly distributed between 0 and $2^{31} - 1$ [PM88, Car90]. The random number produced by *fast_random()* is then scaled[1] to reside in the interval $[0,$ *global_tickets*$-1]$, which will be referred to as the *ticket space*. The scaled random number, *winner*, represents the offset of the winning ticket in the ticket space. The ticket space is then scanned by traversing the client list, accumulating a running ticket *sum* until the winning offset is reached. The client holding the ticket at the winning offset is selected as the winner.

Performing an allocation using the simple list-based lottery algorithm in Figure 3-2 requires $O(n_c)$ time to traverse the list of clients. 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 also be very effective.

For large n_c, a tree-based implementation is more efficient, requiring only $O(\lg n_c)$ operations to perform an allocation. A tree-based implementation would also be more appropriate for a distributed lottery scheduler. Figure 3-3 lists ANSI C code for a tree-based lottery scheduling algorithm. Although many tree-based data structures are possible, a balanced binary tree is used to illustrate the algorithm. Every node has the usual tree links to its parent, left child, and right child, as well as a ticket count. Each leaf node represents an individual client. Each internal node represents the group of clients (leaf nodes) that it covers, and contains their aggregate ticket sum. An allocation is performed by tracing a path from the root of the tree to a leaf. At each level, the child that covers the region of the ticket space which contains the winning ticket is followed. When a leaf node is reached, it is selected as the winning client.

Figure 3-4 illustrates an example tree-based lottery. Eight clients are competing for a resource with a total of 48 tickets. The twenty-fifth ticket is randomly chosen, and a root-to-leaf path is traversed to locate the winning client. Since the winning offset does not appear in the region of the ticket space covered by the root's left child, its right child is followed. The winning offset is adjusted from 25 to 15 to reflect the new subregion of the ticket space that excludes the first ten tickets. At this second level, the adjusted offset of 15 falls within the left child's region of the ticket space. Finally, its right child is followed, with an adjusted winning offset of 3. Since this node is a leaf, it is selected as the winning client.

[1] An exact scaling method would convert the random number from an integer to a floating-point number between 0 and 1, multiply it by *global_tickets*, and then convert the result back to the nearest integer. A more efficient scaling method, used in Figure 3-2, is to simply compute the remainder of the random number modulo *global_tickets*. This method works extremely well under the reasonable assumption that *global_tickets* $\ll 2^{31}$.

```
/* binary tree node */
typedef struct {
  ...
  struct node *left, *right, *parent;
  int tickets;
} *node_t;

/* current resource owner */
node_t current;

/* tree of clients competing for resource */
node_t root;

/* initialize client with specified allocation */
void client_init(node_t c, int tickets)
{
  node_t n;

  /* attach client to tree as leaf */
  tree_insert(root, c);

  /* initialize client state, update ancestor ticket sums */
  c->tickets = tickets;
  for (n = c->parent;
       n != NULL;
       n = n->parent)
    n->tickets += tickets;
}
```

```
/* proportional-share resource allocation */
void allocate()
{
  int winner;
  node_t n;

  /* randomly select winning ticket */
  winner = fast_random() % root->tickets;

  /* traverse root-to-leaf path to find winner */
  for (n = root; !node_is_leaf(n); )
    if (n->left != NULL &&
        n->left->tickets > winner)
      n = n->left;
    else
      {
        /* adjust relative offset for winner */
        n = n->right;
        winner -= n->left->tickets;
      }

  /* use resource */
  current = n;
  use_resource(current);
}
```

Figure 3-3: **Tree-Based Lottery Scheduling Algorithm.** ANSI C code for scheduling a static set of clients using a tree-based lottery. The main data structure is a binary tree of nodes. Each node represents either a client (leaf) or a group of clients and their aggregate ticket sum (internal node). An allocation requires $O(\lg n_c)$ time to locate the winning ticket.

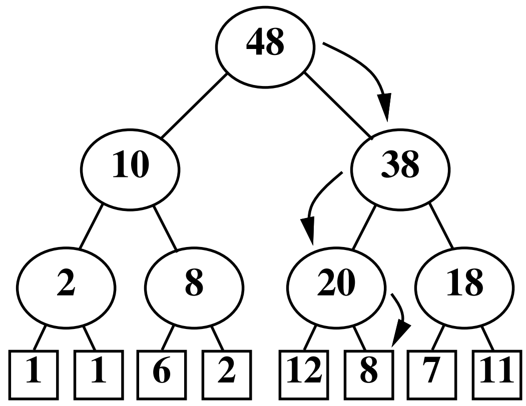

Figure 3-4: **Example Tree-Based Lottery.** Eight clients compete in a tree-based lottery with a total of 48 tickets. Each square leaf node represents a client and its associated ticket allocation. Each round internal node contains the ticket sum for the leaves that it covers. In this example, the winning ticket number is 25, and the winning client is found by traversing the root-to-leaf path indicated by the arrows.

3.1.2 Dynamic Operations

The basic algorithms presented in Figures 3-2 and 3-3 do not support dynamic operations, such as changes in the number of clients competing for a resource, and modifications to client ticket allocations. Fortunately, the use of randomization makes adding such support trivial. Since each random allocation is independent, there is no per-client state to update in response to dynamic changes. Because lottery scheduling is effectively stateless, a great deal of complexity is eliminated. For each allocation, every client is given a fair chance of winning proportional to its share of the total number of tickets. Any dynamic changes are immediately reflected in the next allocation decision, and no special actions are required.

Figure 3-5 lists ANSI C code that trivially extends the basic list-based algorithm to efficiently handle dynamic changes. The time complexity of the *client_modify()*, *client_leave()*, and *client_join()* operations is $O(1)$. Figure 3-6 lists the corresponding extensions for the basic tree-based algorithm. These operations require $O(\lg n_c)$ time to update the ticket sums for each of a client's ancestors. The list-based *client_modify()* operation and the tree-based *node_modify()* operation update global scheduling state only for clients that are actively competing for resources.[2]

[2]The *client_is_active()* predicate can be implemented simply by associating an explicit *active* flag with each client. This flag should be set in *client_join()* and reset in *client_leave()*. An alternative implementation of *client_is_active()* could simply check if the client's list-link fields are NULL. Similar approaches can be employed to define the *node_is_active()* predicate used in the tree-based implementation of *node_modify()*.

```
/* dynamically modify client allocation by delta tickets */
void client_modify(client_t c, int delta)
{
  /* update client tickets */
  c->tickets += delta;

  /* update global ticket sum if active */
  if (client_is_active(c))
    global_tickets += delta;
}
```

```
/* join competition for resource */
void client_join(client_t c)
{
  /* update global ticket sum, link into list */
  global_tickets += c->tickets;
  list_insert(list, c);
}

/* leave competition for resource */
void client_leave(client_t c)
{
  /* update global ticket sum, unlink from list */
  global_tickets -= c->tickets;
  list_remove(list, c);
}
```

Figure 3-5: **Dynamic Operations: List-Based Lottery.** ANSI C code to support dynamic operations for a list-based lottery scheduler. All operations execute in constant time.

```
/* dynamically modify node allocation by delta tickets */
void node_modify(node_t node, int delta)
{
  node_t n;

  /* update node tickets */
  node->tickets += delta;

  /* propagate changes to ancestors if active */
  if (node_is_active(node))
    for (n = node->parent;
         n != NULL;
         n = n->parent)
      n->tickets += delta;
}
```

```
/* join competition for resource */
void client_join(node_t c)
{
  /* add node to tree, update ticket sums */
  tree_insert(root, c);
  node_modify(c->parent, c->tickets);
}

/* leave competition for resource */
void client_leave(node_t c)
{
  /* update ticket sums, remove node from tree */
  node_modify(c->parent, - c->tickets);
  tree_remove(root, c);
}
```

Figure 3-6: **Dynamic Operations: Tree-Based Lottery.** ANSI C code to support dynamic operations for a tree-based lottery scheduler. All operations require $O(\lg n_c)$ time to update ticket sums.

3.1.3 Nonuniform Quanta

With the basic lottery scheduling algorithms presented in Figures 3-2 and 3-3, a client that does not consume its entire allocated quantum will receive less than its entitled share. Similarly, it may be possible for a client's usage to exceed a standard quantum in some situations. For example, under a non-preemptive scheduler, the amount of time that clients hold a resource can vary considerably.

Fractional and variable-size quanta are handled by adjusting a client's ticket allocation to compensate for its nonuniform quantum usage. When a client consumes a fraction f of its allocated time quantum, it is assigned transient *compensation tickets* that alter its overall ticket value by $1/f$ until the client starts its next quantum. This ensures that a client's expected resource consumption, equal to f times its per-lottery win probability p, is adjusted by $1/f$ to match its allocated share. If $f < 1$, then the client will receive positive compensation tickets, inflating its effective ticket allocation. If $f > 1$, then the client will receive negative compensation tickets, deflating its effective allocation.

To demonstrate that compensation tickets have the desired effect, consider a client that owns t of the T tickets competing for a resource. Suppose that when the client next wins the resource lottery, it uses a fraction f of its allocated quantum. The client is then assigned $t/f - t$ transient compensation tickets, changing its overall ticket value to t/f. These compensation tickets persist only until the client wins another allocation.

Without any compensation, the client's expected waiting time until its next allocation would be $T/t - 1$ quanta. Compensation alters both the client's ticket allocation and the total number of tickets competing for the resource. With compensation, the client's expected waiting time becomes $(T + t/f - t)/(t/f) - 1$, which reduces to $fT/t - f$. Measured from the start of its first allocation to the start of its next allocation, the client's expected resource usage is f quanta over a time period consisting of $f + (fT/t - f) = fT/t$ quanta. Thus, the client receives a resource share of $f/(fT/t) = t/T$, as desired.

Note that no assumptions were made regarding the client's resource usage during its second allocation. Compensation tickets produce the correct expected behavior even when f varies dynamically, since the client's waiting time is immediately adjusted after every allocation. A malicious client is therefore unable to boost its resource share by varying f in an attempt to "game" the system.

Figure 3-7 lists ANSI C code for compensating a client that uses *elapsed* resource time units instead of a standard *quantum*, measured in the same time units. The per-client scheduling state is extended to include a new *compensate* field that contains the current number of compensation tickets associated with the client. The *compensate()* operation should be invoked immediately

```c
/* per-client state */
typedef struct {
  ...
  int tickets, compensate;
} *client_t;

/* standard quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);
```

```c
/* compensate client for nonuniform quantum usage */
void compensate(client_t c, int elapsed)
{
  int old, new, net_change;

  /* compute original allocation */
  old = c->tickets - c->compensate;

  /* compute current compensation */
  new = (old * quantum) / elapsed;
  c->compensate = new - old;

  /* compute change, modify effective allocation */
  net_change = new - c->tickets;
  client_modify(c, net_change);
}
```

Figure 3-7: **Compensation Ticket Assignment.** ANSI C code to compensate a client for consuming *elapsed* time units of a resource instead of a standard time slice of *quantum* time units. This code assumes a list-based lottery; a tree-based lottery would simply replace the invocation of *client_modify()* with *node_modify()*.

after every allocation; *compensate(current, elapsed)* should be added to the end of the *allocate()* operation. Compensation tickets are transient, and only persist until the client starts its next quantum. Thus, *compensate()* initially forgets any previous compensation, and computes a new client compensation value based on *elapsed*. The client's *compensate* field is updated, and the overall difference between the previous compensated ticket value and its new one is computed as *net_change*. Finally, the client's ticket allocation is dynamically modified via *client_modify()*.

For example, suppose clients A and B have each been allocated 400 tickets. Client A always consumes its entire quantum, while client B uses only one-fifth of its quantum before yielding the resource. Since both A and B have equal ticket assignments, they are equally likely to win a lottery when both compete for the same resource. However, client B uses only $f = 1/5$ of its allocated time, allowing client A to consume five times as much of the resource, in violation of their 1 : 1 ticket ratio. To remedy this situation, client B is granted 1600 compensation tickets when it yields the resource. When B next competes for the resource, its total funding will be $400/f = 2000$ tickets. Thus, on average B will win the resource 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.

3.2 Multi-Winner Lottery Scheduling

Multi-winner lottery scheduling is a generalization of the basic lottery scheduling technique. Instead of selecting a single winner per lottery, n_w winners are selected, and each winner is granted the use of the resource for one quantum. The set of n_w consecutive quanta allocated by a single multi-winner lottery will be referred to as a *superquantum*. This section presents the basic multi-winner lottery algorithm, followed by a discussion of extensions for dynamic operations and nonuniform quanta.

3.2.1 Basic Algorithm

The multi-winner lottery scheduling algorithm is a hybrid technique with both randomized and deterministic components. The first winner in a superquantum is selected randomly, and the remaining $n_w - 1$ winners are selected deterministically at fixed offsets relative to the first winner. These offsets appear at regular, equally-spaced intervals in the *ticket space* $[0, T - 1]$, where T is the total number of tickets competing for the resource. More formally, the n_w winning offsets are located at $(r + i\frac{T}{n_w}) \bmod T$ in the ticket space, where r is a random number and index $i \in [0, n_w - 1]$ yields the i^{th} winning offset.

Since individual winners within a superquantum are uniformly distributed across the ticket space, multi-winner lotteries directly implement a form of short-term, proportional-share fairness. Because the spacing between winners is T/n_w tickets, a client with t tickets is deterministically guaranteed to receive at least $\lfloor n_w \frac{t}{T} \rfloor$ quanta per superquantum. However, there are no deterministic guarantees for clients with fewer than T/n_w tickets.

An appropriate value for n_w can be computed by choosing the desired level of deterministic guarantees. Larger values of n_w result in better deterministic approximations to specified ticket allocations, reducing the effects of random error. Ensuring that a client deterministically receives at least one quantum per superquantum substantially increases its throughput accuracy and dramatically reduces its response-time variability. Setting $n_w \geq 1/f$ guarantees that all clients entitled to at least a fraction f of the resource will be selected during each superquantum. For example, if deterministic guarantees are required for all clients with resource shares of at least 12.5%, then a value of $n_w \geq 8$ should be used.

Figure 3-8 presents an example multi-winner lottery. Five clients compete for a resource with a total of $T = 20$ tickets. The thirteenth ticket is randomly chosen, resulting in the selection of the third client as the first winner. Since $n_w = 4$, three additional winners are selected in the same superquantum, with relative offsets that are multiples of $T/4 = 5$ tickets. Note that the first client with 10 tickets is guaranteed to receive 2 out of every 4 quanta, and the third client with 5 tickets is guaranteed to receive 1 out of every 4 quanta. The choice of

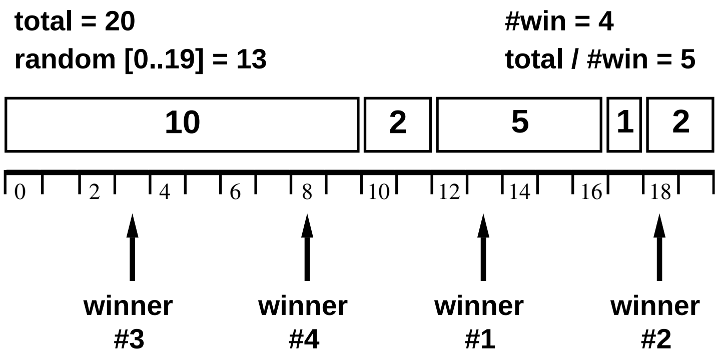

Figure 3-8: **Example Multi-Winner Lottery.** Five clients compete in a four-winner lottery with a total of 20 tickets. The first winner is selected at a randomly-generated offset of 13, and the remaining winners are selected at relative offsets with a deterministic spacing of 5 tickets.

---

the client that receives the remaining quantum is effectively determined by the random number generated for the superquantum.

Although the basic multi-winner lottery mechanism is very simple, the use of a superquantum introduces a few complications. One issue is the ordering of winning clients within a superquantum. The simplest option is to schedule the clients in the order that they are selected. However, this can result in the allocation of several consecutive quanta to clients holding a relatively large number of tickets. While this is desirable in some cases to reduce context-switching overhead, the reduced interleaving also increases response time variability. Another straightforward approach with improved interleaving is to schedule the winning clients using an ordering defined by a fixed or pseudo-random permutation.

Figure 3-9 lists ANSI C code for a list-based multi-winner lottery algorithm that schedules winners within a superquantum using a fixed permuted order. The per-client state and *client_init()* operation are identical to those listed in Figure 3-2. Additional global state is introduced to handle the scheduling of winners within a superquantum. The *intra_schedule* array defines a fixed permutation of winners within a superquantum, such that successive winners are maximally separated from one another in the ticket space. The random offset for the first winner is maintained by *intra_first*, and the deterministic spacing between winners is maintained by *intra_space*. The current intra-superquantum winner number is stored by *intra_count*.

The *allocate()* operation initially checks if a new superquantum should be started by inspecting *intra_count*. When a superquantum is started, a new random winning offset is generated, and a new deterministic inter-winner spacing is computed. These same values are then used for all of the allocations within the superquantum. Each allocation determines the next winner

```
/* per-client state */
typedef struct {
  ...
  int tickets;
} *client_t;

/* winners per superquantum (e.g. 4) */
const int n_winners = 4;

/* current resource owner */
client_t current;

/* list of clients competing for resource */
list_t list;

/* global ticket sum */
int global_tickets = 0;

/* intra-superquantum schedule (e.g. permuted) */
int intra_schedule[] = { 0, 2, 1, 3 };
int intra_first, intra_space;
int intra_count = 0;

/* locate client with winning offset */
client_t find_winner(int winner)
{
  int sum = 0;
  client_t c;

  /* search list to find client with winning offset */
  for (c = list_first(list);
       c != NULL;
       c = list_next(list, c))
    {
      /* update running sum, stop at winner */
      sum += c->tickets;
      if (sum > winner)
        return(c);
    }
}

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* initialize client state, update global sum */
  c->tickets = tickets;
  global_tickets += tickets;

  /* join competition for resource */
  list_insert(list, c);
}

/* proportional-share resource allocation */
void allocate()
{
  int winner;

  /* handle new superquantum */
  if (intra_count == 0)
    {
      /* generate random offset, inter-winner spacing */
      intra_first =
        fast_random() % global_tickets;
      intra_space = global_tickets / n_winners;
    }

  /* select next winner within superquantum */
  winner = intra_first +
    intra_space * intra_schedule[intra_count];

  /* handle ticket-space wrap-around */
  if (winner >= global_tickets)
    winner -= global_tickets;

  /* advance intra-superquantum winner count */
  if (++intra_count == n_winners)
    intra_count = 0;

  /* grant resource to winner for quantum */
  current = find_winner(winner);
  use_resource(current);
}
```

Figure 3-9: **Multi-Winner Lottery Scheduling Algorithm.** ANSI C code for scheduling a static set of clients using a list-based multi-winner lottery. An allocation requires $O(n_c)$ time to search the list of clients for the winning ticket.

by computing its offset within the ticket space. This winning offset is the sum of the initial random offset, *intra_first*, and a deterministic offset based on the relative position of the next winner, *intra_space* $\times$ *intra_sched[intra_count]*. Thus, successive winners within the same superquantum are separated by some multiple of *intra_space* tickets. The implementation of the *find_winner()* operation is identical to the linear search used in Figure 3-2, and is presented as a separate abstraction to highlight the key changes to *allocate()*.

A more efficient version of the code listed in Figure 3-9 can be implemented by selecting all of the superquantum winners during a single scan of the client list. By avoiding a separate pass for each allocation, this optimization would also decrease the cost of performing an allocation by nearly a factor of n_w over ordinary lottery scheduling. The implementation of a tree-based multi-winner lottery would also be very similar to the list-based code. The *find_winner()* function can simply be changed to use the tree-based search employed in Figure 3-3, and references to *global_tickets* can be replaced by the *root* node's *tickets* field.

The multi-winner lottery algorithm is very similar to the *stochastic remainder* technique used in the field of genetic algorithms for randomized population mating and selection [Gol89]. This technique can also be applied to scheduling time-shared resources, although it was not designed for that purpose. Using the same scheduling terminology introduced earlier, for each superquantum consisting of n_w consecutive quanta, the stochastic remainder technique allocates each client $n_w \frac{t}{T}$ quanta, where t is the number of tickets held by that client, and T is the total number of tickets held by all clients. The integer part of this expression is deterministically allocated, and the fractional *remainder* is stochastically allocated by lottery.

For example, consider a superquantum with $n_w = 10$, and two clients, A and B, with a 2 : 1 ticket allocation ratio. Client A receives $\lfloor 10 \times \frac{2}{3} \rfloor = 6$ quanta, and B receives $\lfloor 10 \times \frac{1}{3} \rfloor = 3$ quanta. Thus, A is deterministically guaranteed to receive six quanta out of every ten; B is guaranteed to receive three quanta out of every ten. The remaining quantum is allocated by lottery with probability $(10 \times \frac{2}{3}) - 6 = \frac{2}{3}$ to client A, and $(10 \times \frac{1}{3}) - 3 = \frac{1}{3}$ to client B.

The multi-winner lottery algorithm and the stochastic remainder technique both provide the same deterministic guarantee: a client with t tickets will receive at least $\lfloor n_w \frac{t}{T} \rfloor$ quanta per superquantum. The remaining quanta are allocated stochastically. The stochastic remainder approach uses independent random numbers to perform these allocations, while a multi-winner lottery bases its allocations on a single random number. A multi-winner lottery evenly divides the ticket space into regions, and selects a winner from each region by lottery. This distinction provides several implementation advantages. For example, fewer random numbers are generated; the same random number is effectively reused within a superquantum. Also, fewer expensive arithmetic operations are required. In addition, if n_w is chosen to be a power of two, then all divisions can be replaced with efficient shift operations.

```
/* dynamically modify client allocation by delta tickets */
void client_modify(client_t c, int delta)
{
  /* update client tickets */
  c->tickets += delta;

  /* adjust global state if active */
  if (client_is_active(c))
    {
      /* force start of new superquantum */
      intra_count = 0;

      /* update global ticket sum */
      global_tickets += delta;
    }
}
```

```
/* join competition for resource */
void client_join(client_t c)
{
  /* force start of new superquantum */
  intra_count = 0;

  /* update global ticket sum, link into list */
  global_tickets += c->tickets;
  list_insert(list, c);
}

/* leave competition for resource */
void client_leave(client_t c)
{
  /* force start of new superquantum */
  intra_count = 0;

  /* update global ticket sum, unlink from list */
  global_tickets -= c->tickets;
  list_remove(list, c);
}
```

Figure 3-10: **Dynamic Operations: Multi-Winner Lottery.** ANSI C code to support dynamic operations for a list-based multi-winner lottery scheduler. Each operation terminates the current superquantum. All operations execute in constant time.

3.2.2 Dynamic Operations

The use of a superquantum also complicates operations that dynamically modify the set of competing clients or their relative ticket allocations. For a single-winner lottery, each allocation is independent, and there is no state that must be transformed in response to dynamic changes. For a multi-winner lottery, the current state of the intra-superquantum schedule must be considered.

Randomization can be used to once again sidestep the complexities of dynamic modifications, by scheduling winners within a superquantum in a pseudo-random order. After any dynamic change, the current superquantum is simply prematurely terminated and a new superquantum is started. This same technique can also be used with an intra-superquantum schedule based on a fixed permutation, such as the one listed in Figure 3-9. Since winners are maximally separated in the ticket space, premature termination of a superquantum after w winners have been selected approximates the behavior exhibited by a multi-winner lottery scheduler with $n_w = w$. For example, the first two winners scheduled by the four-winner lottery listed in Figure 3-9 are identical to the winners that would be selected by a two-winner lottery. When n_w and w are perfect powers of two, this approximation will be exact. In other cases, the use of a randomly-generated initial offset still ensures that no systematic bias will develop across superquanta. This is important, because systematic bias could potentially be exploited by clients attempting to cheat the system.

Figure 3-10 lists ANSI C code that trivially extends the basic multi-winner lottery algorithm to handle dynamic changes. The premature termination of a superquantum allows dynamic operations to be supported in a principled manner. However, if dynamic changes occur with high frequency, then the effective superquantum size will be reduced, weakening the deterministic guarantees that it was intended to provide. In the extreme case where a dynamic change occurs after every allocation, this scheme reduces to an ordinary single-winner lottery. I was unable to find other systematic dynamic techniques that work with alternative ordering schemes. In general, the use of a superquantum introduces state that may require complicated transformations to avoid incorrect dynamic behavior.

3.2.3 Nonuniform Quanta

Fractional and variable-size quanta are supported by the same compensation ticket technique described for ordinary lottery scheduling. The code presented for assigning compensation tickets in Figure 3-7 can be used without modification. However, for multi-winner lotteries, the assignment of compensation tickets forces the start of a new superquantum, since the multi-winner version of *client_modify()* terminates the current superquantum. Thus, if clients frequently use nonuniform quantum sizes, the effective superquantum size will be reduced, weakening the deterministic guarantees provided by the multi-winner lottery.

The need to start a new superquantum after every nonuniform quantum can be avoided by using a more complex compensation scheme. Instead of invoking *compensate()* after every allocation, compensation tickets can be assigned after each complete superquantum. This approach requires keeping track of each winner's cumulative allocation count and resource usage over the entire superquantum to determine appropriate compensation values.

3.3 Deterministic Stride Scheduling

Stride scheduling is a deterministic allocation mechanism for time-shared resources. Stride scheduling implements proportional-share control over processor-time and other resources by cross-applying and generalizing elements of rate-based flow control algorithms designed for networks [DKS90, Zha91, ZK91, PG93]. New techniques are introduced to efficiently support dynamic operations, such as modifications to ticket allocations, and changes to the number of clients competing for a resource.

```c
/* per-client state */
typedef struct {
  ...
  int tickets, stride, pass;
} *client_t;

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* queue of clients competing for resource */
queue_t queue;

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* stride is inverse of tickets */
  c->tickets = tickets;
  c->stride = stride1 / tickets;
  c->pass = c->stride;

  /* join competition for resource */
  queue_insert(queue, c);
}
```

```c
/* proportional-share resource allocation */
void allocate()
{
  /* select client with minimum pass value */
  current = queue_remove_min(queue);

  /* use resource for quantum */
  use_resource(current);

  /* compute next pass using stride */
  current->pass += current->stride;
  queue_insert(queue, current);
}
```

Figure 3-11: **Basic Stride Scheduling Algorithm.** ANSI C code for scheduling a static set of clients. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

3.3.1 Basic Algorithm

The core stride scheduling idea is to compute a representation of the time interval, or *stride*, that a client must wait between successive allocations. The client with the smallest stride will be scheduled most frequently. A client with half the stride of another will execute twice as quickly; a client with double the stride of another will execute twice as slowly. Strides are represented in virtual time units called *passes*, instead of units of real time such as seconds.

Three state variables are associated with each client: *tickets*, *stride*, and *pass*. The *tickets* field specifies the client's resource allocation, relative to other clients. The *stride* field is inversely proportional to *tickets*, and represents the interval between selections, measured in passes. The *pass* field represents the virtual time index for the client's next selection. Performing a resource allocation is very simple: the client with the minimum *pass* is selected, and its *pass* is advanced by its *stride*. If more than one client has the same minimum pass value, then any of them may be selected. A reasonable deterministic approach is to use a consistent ordering to break ties, such as one defined by unique client identifiers.

The only source of relative error under stride scheduling is due to quantization. Thus, the the relative error for any pair of clients is never greater than *one*, independent of n_a. However, for skewed ticket distributions it is still possible for a client to have $O(n_c)$ absolute error, where n_c is the number of clients. Nevertheless, stride scheduling is considerably more accurate than lottery scheduling, since its error does not grow with the number of allocations.

Figure 3-11 lists ANSI C code for the basic stride scheduling algorithm. For simplicity, a static set of clients with fixed ticket assignments is assumed. These restrictions will be relaxed in subsequent sections to permit more dynamic behavior. The stride scheduling state for each client must be initialized via *client_init()* before any allocations are performed by *allocate()*. To accurately represent *stride* as the reciprocal of *tickets*, a floating-point representation could be used. A more efficient alternative is presented that uses a high-precision fixed-point integer representation. This is easily implemented by multiplying the inverted ticket value by a large integer constant. This constant will be referred to as *stride*$_1$, since it represents the stride corresponding to the minimum ticket allocation of one.[3]

The cost of performing an allocation depends on the data structure used to implement the client queue. A priority queue can be used to implement *queue_remove_min()* and other queue operations in $O(\lg n_c)$ time or better, where n_c is the number of clients [CLR90, Tho95]. A skip list could also provide expected $O(\lg n_c)$ time queue operations with low constant overhead [Pug90]. For small n_c or heavily skewed ticket distributions, a simple sorted list is likely to be most efficient in practice.

Figure 3-12 illustrates an example of stride scheduling. Three clients, A, B, and C, are competing for a time-shared resource with a 3 : 2 : 1 ticket ratio. For simplicity, a convenient *stride*$_1$ = 6 is used instead of a large number, yielding respective strides of 2, 3, and 6. The pass value of each client is plotted as a function of time. For each quantum, the client with the minimum pass value is selected, and its pass is advanced by its stride. Ties are broken using the arbitrary but consistent client ordering A, B, C. The sequence of allocations produced by stride scheduling in Figure 3-12 exhibits precise periodic behavior: A, B, A, A, B, C.

3.3.2 Dynamic Operations

The basic stride scheduling algorithm presented in Figure 3-11 does not support dynamic changes in the number of clients competing for a resource. When clients are allowed to join and leave at any time, their state must be appropriately modified. Figure 3-13 extends the basic algorithm to efficiently handle dynamic changes to the set of active clients. The code listed in Figure 3-13 also supports nonuniform quanta; this issue will be discussed in Section 3.3.3.

[3] Section 5.2.1 discusses the representation of strides in more detail.

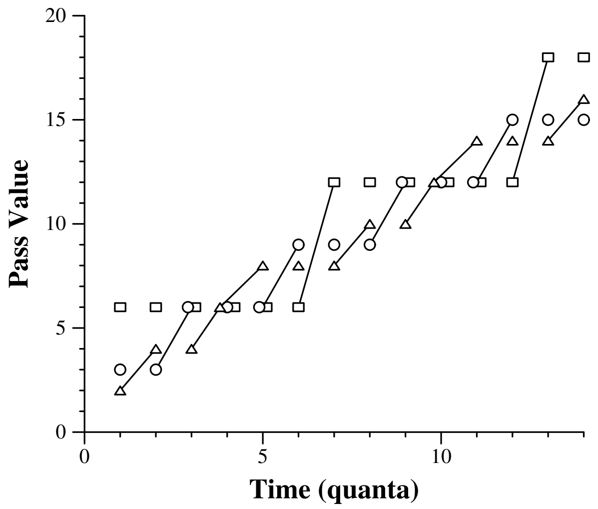

Figure 3-12: **Stride Scheduling Example.** Clients A (triangles), B (circles), and C (squares) have a 3 : 2 : 1 ticket ratio. In this example, stride_1 = 6, yielding respective strides of 2, 3, and 6. For each quantum, the client with the minimum pass value is selected, and its pass is advanced by its stride.

A key extension is the addition of global variables that maintain aggregate information about the set of active clients. The *global_tickets* variable contains the total ticket sum for all active clients. The *global_pass* variable maintains the "current" pass for the scheduler. The *global_pass* advances at the rate of *global_stride* per quantum, where $\mathit{global\_stride} = \mathit{stride}_1 / \mathit{global\_tickets}$. Conceptually, the *global_pass* continuously advances at a smooth rate. This is implemented by invoking the *global_pass_update()* routine whenever the *global_pass* value is needed.[4]

A state variable is also associated with each client to store the remaining portion of its stride when a dynamic change occurs. The *remain* field represents the number of passes that are left before a client's next selection. When a client leaves the system, *remain* is computed as the difference between the client's *pass* and the *global_pass*. When a client rejoins the system, its *pass* value is recomputed by adding its *remain* value to the *global_pass*.

This mechanism handles situations involving either positive or negative error between the specified and actual number of allocations. If $\mathit{remain} < \mathit{stride}$, then the client is effectively given credit when it rejoins for having previously waited for part of its stride without receiving

[4]Due to the use of a fixed-point integer representation for strides, small quantization errors may accumulate slowly, causing *global_pass* to drift away from client pass values over a long period of time. This is unlikely to be a practical problem, since client pass values are recomputed using *global_pass* each time they leave and rejoin the system. However, this problem can be avoided by infrequently resetting *global_pass* to the minimum pass value for the set of active clients.

```c
/* per-client state */
typedef struct {
  ...
  int tickets, stride, pass, remain;
} *client_t;

/* quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* queue of clients competing for resource */
queue_t queue;

/* global aggregate tickets, stride, pass */
int global_tickets, global_stride, global_pass;

/* update global pass based on elapsed real time */
void global_pass_update(void)
{
  static int last_update = 0;
  int elapsed;

  /* compute elapsed time, advance last_update */
  elapsed = time() - last_update;
  last_update += elapsed;

  /* advance global pass by quantum-adjusted stride */
  global_pass +=
    (global_stride * elapsed) / quantum;
}

/* update global tickets and stride to reflect change */
void global_tickets_update(int delta)
{
  global_tickets += delta;
  global_stride = stride1 / global_tickets;
}

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* stride is inverse of tickets, whole stride remains */
  c->tickets = tickets;
  c->stride = stride1 / tickets;
  c->remain = c->stride;
}

/* join competition for resource */
void client_join(client_t c)
{
  /* compute pass for next allocation */
  global_pass_update();
  c->pass = global_pass + c->remain;

  /* add to queue */
  global_tickets_update(c->tickets);
  queue_insert(queue, c);
}

/* leave competition for resource */
void client_leave(client_t c)
{
  /* compute remainder of current stride */
  global_pass_update();
  c->remain = c->pass - global_pass;

  /* remove from queue */
  global_tickets_update(-c->tickets);
  queue_remove(queue, c);
}

/* proportional-share resource allocation */
void allocate()
{
  int elapsed;

  /* select client with minimum pass value */
  current = queue_remove_min(queue);

  /* use resource, measuring elapsed real time */
  elapsed = use_resource(current);

  /* compute next pass using quantum-adjusted stride */
  current->pass +=
    (current->stride * elapsed) / quantum;
  queue_insert(queue, current);
}
```

Figure 3-13: **Dynamic Stride Scheduling Algorithm.** ANSI C code for stride scheduling operations, including support for joining, leaving, and nonuniform quanta. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

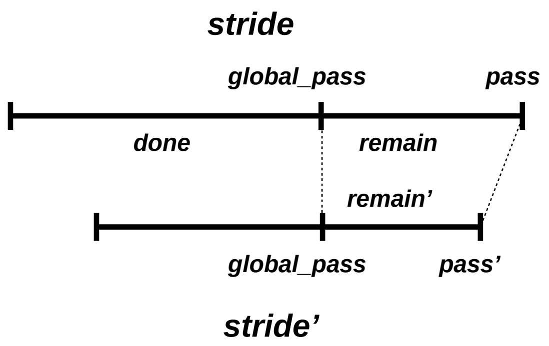

Figure 3-14: **Stride Scheduling Allocation Change.** Modifying a client's allocation from *tickets* to *tickets*$'$ requires only a constant-time recomputation of its *stride* and *pass*. The new *stride*$'$ is inversely proportional to *tickets*$'$. The new *pass*$'$ is determined by scaling *remain*, the remaining portion of the the current *stride*, by *stride*$'$ / *stride*.

---

a quantum. If *remain* $>$ *stride*, then the client is effectively penalized when it rejoins for having previously received a quantum without waiting for its entire stride.[5] This approach implicitly assumes that a partial quantum now is equivalent to a partial quantum later. In general, this is a reasonable assumption, and resembles the treatment of nonuniform quanta that will be presented in Section 3.3.3. However, it may not be appropriate if the total number of tickets competing for a resource varies significantly between the time that a client leaves and rejoins the system.

The time complexity for both the *client_leave()* and *client_join()* operations is $O(\lg n_c)$, where n_c is the number of clients. These operations are efficient because the stride scheduling state associated with distinct clients is completely independent; a change to one client does not require updates to any other clients. The $O(\lg n_c)$ cost results from the need to perform queue manipulations.

Additional support is needed to dynamically modify client ticket allocations. Figure 3-14 illustrates a dynamic allocation change, and Figure 3-15 lists ANSI C code for dynamically changing a client's ticket allocation. When a client's allocation is dynamically changed from *tickets* to *tickets*$'$, its *stride* and *pass* values must be recomputed. The new *stride*$'$ is computed as usual, inversely proportional to *tickets*$'$. To compute the new *pass*$'$, the remaining portion of the client's current *stride*, denoted by *remain*, is adjusted to reflect the new *stride*$'$. This is accomplished by scaling *remain* by *stride*$'$ / *stride*. In Figure 3-14, the client's ticket allocation

[5]Several interesting alternatives could also be implemented. For example, a client could be given credit for some or all of the passes that elapse while it is inactive.

```
/* dynamically modify client ticket allocation by delta tickets */
void client_modify(client_t c, int delta)
{
  int tickets, stride, remain;
  bool_t active;

  /* check if client actively competing for resource */
  active = client_is_active(c);

  /* leave queue for resource */
  if (active)
    client_leave(c);

  /* compute new tickets, stride */
  tickets = c->tickets + delta;
  stride = stride1 / tickets;

  /* scale remaining passes to reflect change in stride */
  remain = (c->remain * stride) / c->stride;

  /* update client state */
  c->tickets = tickets;
  c->stride = stride;
  c->remain = remain;

  /* rejoin queue for resource */
  if (active)
    client_join(c);
}
```

Figure 3-15: **Dynamic Ticket Modification: Stride Scheduling.** ANSI C code for dynamic modifications to client ticket allocations under stride scheduling. The *client_modify()* operation requires $O(\lg n_c)$ to perform appropriate queue manipulations.

is increased, so *pass* is decreased, compressing the time remaining until the client is next selected. If its allocation had decreased, then *pass* would have increased, expanding the time remaining until the client is next selected.

The *client_modify()* operation requires $O(\lg n_c)$ time, where n_c is the number of clients. As with dynamic changes to the number of clients, ticket allocation changes are efficient because the stride scheduling state associated with distinct clients is completely independent; the dominant cost is due to queue manipulations.

3.3.3 Nonuniform Quanta

With the basic stride scheduling algorithm presented in Figure 3-11, a client that does not consume its entire allocated quantum will receive less than its entitled share of a resource. Similarly, it may be possible for a client's usage to exceed a standard quantum in some situations. For example, under a non-preemptive scheduler, client run lengths can vary considerably.

Fortunately, fractional and variable-size quanta can easily be accommodated. When a client consumes a fraction f of its allocated time quantum, its *pass* should be advanced by $f \times$ *stride* instead of *stride*. If $f < 1$, then the client's *pass* will be increased less, and it will be scheduled sooner. If $f > 1$, then the client's *pass* will be increased more, and it will be scheduled later. The extended code listed in Figure 3-13 supports nonuniform quanta by effectively computing f as the *elapsed* resource usage time divided by a standard *quantum* in the same time units.

Another extension would permit clients to specify the quantum size that they require.[6] This could be implemented by associating an additional $quantum_c$ field with each client, and scaling each client's stride field by $quantum_c$ / *quantum*. Deviations from a client's specified quantum would still be handled as described above, with f redefined as the *elapsed* resource usage divided by the client-specific $quantum_c$.

3.4 Hierarchical Stride Scheduling

Stride scheduling guarantees that the *relative* throughput error for any pair of clients never exceeds a single quantum. However, depending on the distribution of tickets to clients, a large $O(n_c)$ *absolute* throughput error is still possible, where n_c is the number of clients.

For example, consider a set of 101 clients with a 100 : 1 : . . . : 1 ticket allocation. A schedule that minimizes absolute error and response time variability would alternate the 100-ticket client with each of the single-ticket clients. However, the standard stride algorithm schedules the

[6] Yet another alternative would be to allow each client to specify its scheduling period. Since a client's period and quantum are related by its relative resource share, specifying one quantity yields the other.

clients in order, with the 100-ticket client receiving 100 quanta before any other client receives a single quantum. Thus, after 100 allocations, the intended allocation for the 100-ticket client is 50, while its actual allocation is 100, yielding a large absolute error of 50 quanta. Similar rate-based flow control algorithms designed for networks [DKS90, Zha91, ZK91, PG93] also exhibit this undesirable behavior.

This section describes a novel hierarchical variant of stride scheduling that limits the absolute throughput error of any client to $O(\lg n_c)$ quanta. For the 101-client example described above, hierarchical stride scheduler simulations produced a maximum absolute error of only 4.5. The hierarchical algorithm also significantly reduces response time variability by aggregating clients to improve interleaving. Since it is common for systems to consist of a small number of high-throughput clients together with a large number of low-throughput clients, hierarchical stride scheduling represents a practical improvement over previous work.

3.4.1 Basic Algorithm

Hierarchical stride scheduling is essentially a recursive application of the basic stride scheduling algorithm. Individual clients are combined into groups with larger aggregate ticket allocations, and correspondingly smaller strides. An allocation is performed by invoking the normal stride scheduling algorithm first among groups, and then among individual clients within groups.

Although many different groupings are possible, a balanced binary tree of groups is considered. Each leaf node represents an individual client. Each internal node represents the group of clients (leaf nodes) that it covers, and contains their aggregate ticket, stride, and pass values. Thus, for an internal node, *tickets* is the total ticket sum for all of the clients that it covers, and *stride* = *stride*$_1$ / *tickets*. The *pass* value for an internal node is updated whenever the pass value for any of the clients that it covers is modified.

Figure 3-16 presents ANSI C code for the basic hierarchical stride scheduling algorithm. This code also supports nonuniform quanta, which will be discussed in Section 3.4.3. Each node has the normal tickets, stride, and pass scheduling state, as well as the usual tree links to its parent, left child, and right child. An allocation is performed by tracing a path from the root of the tree to a leaf, choosing the child with the smaller pass value at each level via *node_choose_child()*. Once the selected client has finished using the resource, its pass value is updated to reflect its usage. The client update is identical to that used in the dynamic stride algorithm that supports nonuniform quanta, listed in Figure 3-13. However, the hierarchical scheduler requires additional updates to each of the client's ancestors, following the leaf-to-root path formed by successive parent links.

```c
/* binary tree node */
typedef struct node {
  ...
  struct node *left, *right, *parent;
  int tickets, stride, pass;
} *node_t;

/* quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
node_t current;

/* tree of clients competing for resource */
node_t root;

/* select child of internal node to follow */
node_t node_choose_child(node_t n)
{
  /* no choice if only one child */
  if (n->left == NULL)
    return(n->right);
  if (n->right == NULL)
    return(n->left);

  /* choose child with smaller pass */
  if (n->left->pass < n->right->pass)
    return(n->left);
  else
    return(n->right);
}
```

```c
/* proportional-share resource allocation */
void allocate()
{
  int elapsed;
  node_t n;

  /* traverse root-to-leaf path following min pass */
  for (n = root; !node_is_leaf(n); )
    n = node_choose_child(n);

  /* use resource, measuring elapsed real time */
  current = n;
  elapsed = use_resource(current);

  /* update pass for each ancestor using its stride */
  for (n = current; n != NULL; n = n->parent)
    n->pass +=
      (n->stride * elapsed) / quantum;
}
```

Figure 3-16: **Hierarchical Stride Scheduling Algorithm.** ANSI C code for hierarchical stride scheduling with a static set of clients, including support for nonuniform quanta. The main data structure is a binary tree of nodes. Each node represents either a client (leaf) or a group (internal node) that summarizes aggregate information.

```
/* dynamically modify node allocation by delta tickets */
void node_modify(node_t n, int delta)
{
  int old_stride, remain;

  /* compute new tickets, stride */
  old_stride = n->stride;
  n->tickets += delta;
  n->stride = stride1 / n->tickets;

  /* done when reach root */
  if (n == root)
    return;

  /* simply scale stored remain value if inactive */
  if (!node_is_active(n))
    {
      n->remain = (n->remain * n->stride) / old_stride;
      return;
    }

  /* scale remaining passes to reflect change in stride */
  remain = n->pass - root->pass;
  remain = (remain * n->stride) / old_stride;
  n->pass = root->pass + remain;

  /* propagate change to ancestors */
  node_modify(n->parent, delta);
}
```

Figure 3-17: **Dynamic Ticket Modification: Hierarchical Stride Scheduling.** ANSI C code for dynamic modifications to client ticket allocations under hierarchical stride scheduling. A modification requires $O(\lg n_c)$ time to propagate changes.

Each client allocation can be viewed as a series of pairwise allocations among groups of clients at each level in the tree. The maximum error for each pairwise allocation is 1, and in the worst case, error can accumulate at each level. Thus, the maximum absolute error for a series of tree-based allocations is the height of the tree, which is $\lceil \lg n_c \rceil$, where n_c is the number of clients. Since the error for a pairwise A : B ratio is minimized when $A = B$, absolute error can be further reduced by carefully choosing client leaf positions to better balance the tree based on the number of tickets at each node.

3.4.2 Dynamic Operations

Extending the basic hierarchical stride algorithm to support dynamic modifications requires a careful consideration of the impact that changes have at each level in the tree. Figure 3-17 lists ANSI C code for performing a ticket modification that works for both clients and internal nodes. Changes to client ticket allocations essentially follow the same scaling and update rules used for normal stride scheduling, listed in Figure 3-15. The hierarchical scheduler requires additional updates to each of the client's ancestors, following the leaf-to-root path formed by successive parent links. Note that the *pass* value of the *root* node used in Figure 3-17 effectively takes the place of the *global_pass* variable used in Figure 3-15; both represent the aggregate global scheduler pass.[7]

Operations that allow clients to dynamically join or leave the system must also account for the effects of changes at each level in the tree. Under hierarchical stride scheduling, the state of each node that covers a client is partially based on that client's state. By the time that a client dynamically leaves the system, it may have accumulated $O(\lg n_c)$ absolute error. Without adjustments to compensate for this error, a client that leaves after receiving too few quanta unfairly increases the allocation granted to other clients in the same subtree. Similarly, a client that leaves after receiving too many quanta unfairly decreases the allocation granted to other clients in the same subtree.

A general "undo" and "redo" strategy is used to avoid these problems. Any bias introduced by a client on its ancestors is eliminated when it leaves the system. When the client rejoins the system, symmetric adjustments are made to reconstruct the appropriate bias in order to correctly influence future scheduling decisions.

Figure 3-18 lists ANSI C code that implements support for dynamic client participation. As with ordinary stride scheduling, an additional *remain* field is associated with each client to store the remaining portion of its stride when it leaves the system. If *remain* $<$ *stride*, then the client should be credited for quanta that it was entitled to receive. If *remain* $>$ *stride*, then the client should be penalized when it rejoins the system for having previously received quanta ahead of schedule. As mentioned earlier in the context of dynamic stride scheduling, this approach makes the implicit assumption that a quantum now is equivalent to a quantum later.

When a client leaves the system, *remain* is computed as the difference between its *pass* and the global pass, represented by the *pass* value of the *root* node. When the client rejoins the system, this *remain* value is used to recompute the client's *pass* value. For ordinary stride scheduling, no other special actions are required, because the scheduling state associated with distinct clients is completely independent. However, under hierarchical stride scheduling, the state of each node that covers a client is partially based on that client's state. When a client leaves the system via *client_leave()*, any residual impact on its ancestors must be eliminated. This is implemented by performing a "pseudo-allocation" to erase the effects of client error by updating the client's ancestors as if an actual corrective allocation had been given to the client.

[7] Changes that do not occur on exact quantum boundaries should first update the *root pass* value based on the elapsed real time since its last update. This update would resemble the operation of *global_pass_update()* for ordinary stride scheduling, listed in Figure 3-13.

```
/* binary tree node */
typedef struct node {
  ...
  struct node *left, *right, *parent;
  int tickets, stride, pass, remain;
} *node_t;

/* standard quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* compute entitled resource time for client */
int client_entitled(node_t c)
{
  int completed, entitled;

  /* compute completed passes */
  completed = c->stride - c->remain;

  /* convert completed passes into entitled time */
  entitled =
    (completed * quantum) / c->stride;
  return(entitled);
}

/* pretend that elapsed time units were allocated to node */
void pseudo_allocate(node_t node, int elapsed)
{
  node_t n;

  /* update node, propagate changes to ancestors */
  for (n = node; n != root; n = n->parent)
    n->pass +=
      (n->stride * elapsed) / quantum;
}

/* join competition for resource */
void client_join(node_t c)
{
  /* add node to tree */
  tree_insert(root, c)

  /* perform update to reflect ticket gain */
  node_modify(c->parent, c->tickets);

  /* ``redo'' any existing client error */
  pseudo_allocate(c->parent,
                  - client_entitled(c));

  /* compute pass for next allocation */
  c->pass = root->pass + c->remain;
}

/* leave competition for resource */
void client_leave(node_t c)
{
  /* compute passes remaining */
  c->remain = c->pass - root->pass;

  /* ``undo'' any existing client error */
  pseudo_allocate(c->parent,
                  client_entitled(c));

  /* perform update to reflect ticket loss */
  node_modify(c->parent, - c->tickets);

  /* remove client from tree */
  tree_remove(root, c);
}
```

Figure 3-18: **Dynamic Client Participation: Hierarchical Stride Scheduling.** ANSI C code to support dynamic client participation under hierarchical stride scheduling. The *client_join()* and *client_leave()* operations require $O(\lg n_c)$ time to propagate updates.

This pseudo-allocation is intended to correct any outstanding client error, so the quantum size used for the pseudo-allocation must equal the amount of resource time the client is actually entitled to receive. This value is determined by *client_entitled()*, which returns a resource entitlement measured in the same real time units as *quantum*. A client's entitlement is based on the number of passes remaining before it is due to be selected. If the client's entitlement is positive, then it is currently owed time by the system; if it is negative, then the client owes time to the system.

After the pseudo-allocation corrects for any existing client error, *node_modify()* is invoked to ensure that future updates reflect the overall decrease in tickets due to the client leaving the system. As with any ticket modification, this change propagates to each of the client's ancestors. Finally, a call to *tree_remove()* deactivates the client by removing it from the tree.

When a client rejoins the system via *client_join()*, the inverse operations are performed. First, *tree_insert()* activates the client by adding it to the tree. Next, *node_modify()* is invoked to reflect the overall increase in tickets due to the client joining the system. Finally, the client's new ancestors are updated to reflect its net entitlement by invoking *pseudo_allocate()*. Note that the implementation of *client_join()* is completely symmetric to *client_leave()*. As expected, successive *client_leave()* and *client_join()* operations to the same client leaf position effectively undo one another.

3.4.3 Nonuniform Quanta

Fractional and variable-size quanta are handled in a manner that is nearly identical to their treatment under ordinary stride scheduling. The basic hierarchical stride algorithm listed in Figure 3-16 includes support for nonuniform quanta. When a client uses a fraction f of its allocated quantum, its *pass* is advanced by $f \times$ *stride* instead of *stride*. The same scaling factor f is used when advancing the *pass* values associated with each of the client's ancestors during an allocation.

3.4.4 Huffman Trees

As noted in Section 3.4.1, many different hierarchical groupings of clients are possible. A height-balanced binary tree permits efficient $O(\lg n_c)$ scheduling operations while achieving a $\lceil \lg n_c \rceil$ bound on absolute error. An interesting alternative is to construct a tree with *Huffman's algorithm* [Huf52, CLR90], using client ticket values as frequency counts.[8] Huffman encoding is typically used to find optimal variable-length codes for compressing files or messages.

[8] Thanks to Bill Dally for suggesting this approach.

In the context of hierarchical stride scheduling, a Huffman tree explicitly minimizes the cost of performing an allocation. The same basic allocation operation presented in Figure 3-16 can also be used with Huffman trees. In a Huffman tree, clients with high ticket values are located near the root of the tree, while clients with low ticket values end up farther down in the tree structure. Thus, clients that receive the most allocations require very short root-to-leaf traversals, and clients that receive allocations infrequently have longer root-to-leaf paths.

The Huffman tree structure also ensures that worst-case absolute error is smallest for the clients with the largest ticket values, since maximum absolute error is directly related to the depth of the client in the tree. In Section 4.1.4, it will be demonstrated that response-time variability also increases with tree depth, so a Huffman tree also minimizes response-time variability for large clients. However, $O(n_c)$ absolute error is still possible for small clients, since the height of the tree may be $O(n_c)$ for highly skewed ticket distributions. Allocations to small clients may also exhibit high response-time variability for the same reason.

While a height-balanced tree provides uniform performance bounds and identical allocation costs for all clients, a Huffman tree provides better guarantees and lower allocation costs for clients with larger ticket values, at the expense of clients with smaller ticket values. A detailed analysis of the precise effects of various hierarchical structures is an interesting topic for future research. In addition to bounds on worst-case behavior, more work is needed to better understand the average-case behavior associated with both height-balanced binary trees and Huffman trees.

Huffman trees appear to be a good choice for static environments, since the tree structure remains fixed. However, dynamic operations that modify client ticket values or the set of active clients present some challenging problems. Although the dynamic operations listed in Figures 3-17 and 3-18 will correctly implement proportional sharing with Huffman trees, they do not maintain the invariants that characterize Huffman trees. For example, if a client's ticket allocation is changed from a very small to a very large value, numerous node interchanges and updates are necessary to move the client to a higher location in the tree.

Dynamic Huffman codes have been studied in the context of adaptive compression schemes for communication channels [Vit87]. Similar techniques may be useful for dynamic manipulations of Huffman trees for hierarchical stride scheduling. However, these techniques commonly assume that the values of character frequencies (client ticket values in the case of scheduling) change only incrementally as messages are processed dynamically. Of course, there is no compelling reason that the tree structure used for hierarchical stride scheduling must always remain a strict Huffman tree. Relaxing this constraint may yield algorithms that provide near-optimal allocation costs with acceptable overhead for dynamic operations.

3.5 Framework Implementation

This section explains how the various proportional-share mechanisms presented in this chapter can be used to implement the general resource management framework described in Chapter 2. Implementations of ticket transfers, ticket inflation, and ticket currencies are presented in terms of low-level dynamic operations that have already been defined, such as *client_modify()*. Since primitive dynamic operations have been described for lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling, any of these mechanisms can be used as a substrate for the general framework.

3.5.1 Tickets

A *ticket* is a first-class object that abstractly encapsulates relative resource rights. In the descriptions of the basic mechanisms, all of the tickets associated with a client are compactly represented by a single integer. A complete resource management framework implementation is likely to use a more flexible representation, in which tickets are protected system-level objects that can be explicitly created, destroyed, and transferred between clients and currencies. Despite the use of a more sophisticated ticket representation, the resource rights currently specified by a set of tickets can always be converted into a single integer value, expressed in base units. However, the frequency of such conversions may differ between implementations.

3.5.2 Ticket Transfers

A *ticket transfer* is an explicit transfer of tickets from one client to another. A transfer of t tickets from client A to client B essentially consists of two dynamic ticket modifications. These modifications can be implemented by invoking *client_modify(A, $-t$)* and *client_modify(B, t)*.

For lottery scheduling, the *client_modify()* operations simply change the underlying ticket allocations associated with clients A and B, and update appropriate ticket sums. Under multi-winner lottery scheduling, a ticket transfer also terminates the current superquantum. When A transfers tickets to B under stride scheduling, A's stride and pass will increase, while B's stride and pass will decrease. A slight complication arises for complete ticket transfers; *i.e.*, when A transfers its entire ticket allocation to B. In this case, A's adjusted ticket value is zero, leading to an adjusted stride of infinity (division by zero). This problem can be circumvented by treating a complete transfer from A to B as an update of client B via *client_modify(B, A.tickets)*, and a suspension of client A via *client_leave(A)*. This effectively stores A's *remain* value at the time of the transfer, and defers the computation of its *stride* and *pass* values until it once again receives a non-zero ticket allocation. The same technique can be used to implement ticket transfers for hierarchical stride scheduling.

3.5.3 Ticket Inflation and Deflation

Ticket inflation and *ticket deflation* are alternatives to explicit ticket transfers that alter resource rights by manipulating the overall supply of tickets. An instance of inflation or deflation simply requires a single dynamic modification to a client. If t new tickets are created for client A, then the resulting inflation is implemented via *client_modify(A, t)*. Similarly, if t of A's existing tickets are destroyed, the resulting deflation is implemented via *client_modify(A, $-t$)*.

For lottery scheduling, inflation and deflation simply change the underlying ticket allocation associated with a client. In the case of multi-winner lotteries, ticket inflation and deflation also terminate the current superquantum. For stride scheduling, ticket inflation causes a client's stride and pass to decrease; deflation causes its stride and pass to increase. The effect is the same under hierarchical stride scheduling; similar updates are also applied to each internal node that covers the client.

3.5.4 Ticket Currencies

A *ticket currency* defines a resource management abstraction barrier that contains the effects of ticket inflation in a modular way. Tickets are denominated in currencies, allowing resource rights to be expressed in units that are local to each logical module. The effects of inflation are locally contained by effectively maintaining an *exchange rate* between each local currency and a common *base* currency that is conserved. There are several different implementation strategies for currencies.

One *eager* implementation strategy is to always immediately convert ticket values denominated in arbitrary currencies into units of the common base currency. Any changes to the value of a currency would then require dynamic modifications, via *client_modify()*, to all clients holding tickets denominated in that currency, or one derived from it. An important exception is that changes to the number of tickets in the base currency do not require any modifications to client state. This is because all client scheduling state is computed from ticket values expressed in base units, and the state associated with distinct clients is independent. Thus, the scope of any changes in currency values is limited to exactly those clients which are affected. Since currencies are used to group and isolate logical sets of clients, the impact of currency fluctuations will typically be very localized.

An alternative *lazy* implementation strategy defers the computation of ticket values until they are actually needed. For example, consider a list-based lottery scheduler that is implemented for a system with a fixed number of base tickets. Since only a portion of the ticket space is traversed during an allocation, only those clients that are actually examined need to have their tickets converted into base units. A lazy implementation exploits this fact to defer computing the effects of dynamic changes that result from ticket transfers and inflation[9]. The efficiency of such an approach depends on the relative frequencies of dynamic operations and allocations, as well as the distribution of tickets to clients. Lazy implementations may also benefit by caching ticket and currency values to accelerate conversions.

Various optimizations are also possible. For example, in some systems it may be acceptable to temporarily delay the effect of dynamic changes. If delays are large compared to the average allocation granularity, then performance may be improved by batching changes, such as modifications to currency values. A related optimization is to maintain exchange rates that are only approximately correct and loosely consistent. For example, updates to currency values could be deferred until significant changes accumulate. System-enforced limits could even be placed on the allowed rate of inflation and deflation, avoiding large, rapid fluctuations in currency values. Such optimizations would be particularly useful for distributed scheduler implementations, since communication would be relatively expensive.

[9]Many scheduling operations depend upon accurate maintenance of the total number of active base tickets. Inflation or deflation of the base currency would require immediate work to reflect changes in the overall number of base tickets.

Chapter 4

Performance Results

This chapter presents results that quantify the performance of lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. Basic analytical results are initially introduced to serve as a guide to the behavior of the core scheduling algorithms. Quantitative results obtained from simulation experiments are then presented to further evaluate the scheduling mechanisms in both static and dynamic environments. Many graphical presentations of simulation data are included to facilitate detailed comparisons between the various mechanisms.

4.1 Basic Analysis

In general, there are n_c clients competing for a resource, and each client c_i has t_i tickets, for a total of $T = \sum_{i=1}^{n_c} t_i$ tickets. As described in Chapter 3, the throughput accuracy for each client is quantified by measuring the difference between its specified allocation and the allocation that it actually receives. After n_a consecutive allocations, the *specified* allocation for client c_i is $n_a t_i / T$. A client's *absolute error* is defined as the absolute value of the difference between its specified and actual number of allocations. The pairwise *relative error* between clients c_i and c_j is defined as the absolute error for the subsystem containing only c_i and c_j, where $T = t_i + t_j$, and n_a is the total number of allocations received by both clients.

The response time for each client is measured as the elapsed time from its completion of one quantum, up to and including its completion of another. The response-time variability associated with a client is quantified by the spread of its response-time distribution. The range of this spread is given by its minimum and maximum response times. The response-time distribution can also be characterized by its mean, μ, and its standard deviation, σ. Another useful metric that normalizes variability is the dimensionless coefficient of variation, σ/μ.

The rest of this section presents some basic analytical results for lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical scheduling. Each mechanism is analyzed in terms of both throughput accuracy and response-time variability.

4.1.1 Lottery Scheduling

Lottery scheduling is a randomized algorithm, and can be easily analyzed using well-known results from probability and statistics [Tri82]. The number of lotteries won by a client has a binomial distribution. The probability that client c_i will win a particular lottery is simply $p_i = t_i/T$. After n_a identical allocations, the expected number of wins w_i for client c_i is $E[w_i] = n_a\ p_i$, with variance $\sigma^2_{w_i} = n_a p_i(1 - p_i)$. Thus, the expected throughput error for a client is $O(\sqrt{n_a})$. Since error increases slowly with n_a, throughput accuracy steadily improves when error is measured as a percentage of n_a. Nevertheless, the absolute value of the error can still grow without bound.

The response time for a client has a geometric distribution. The expected number of lotteries l_i that client c_i must wait before completing its first win is $E[l_i] = 1/p_i$, with variance $\sigma^2_{l_i} = (1 - p_i)/p_i^2$. The coefficient of variation is $\sigma_{l_i}/E[l_i] = \sqrt{1 - p_i}$. Thus, the response-time variability for client c_i depends only on its relative share of tickets, $p_i = t_i/T$. When p_i is large, the coefficient of variation is small, as desired. However, when p_i is small, the coefficient of variation approaches one, indicating that response-time variability is extremely high for low-throughput clients.

4.1.2 Multi-Winner Lottery Scheduling

Multi-winner lottery scheduling is a hybrid scheme with both randomized and deterministic components. A multi-winner lottery selects n_w winners per lottery; the n_w consecutive quanta allocated by a lottery are referred to as a superquantum. A multi-winner lottery with n_w winners can be analyzed as n_w separate lotteries, each of which independently selects a winner from an equal-size region of the ticket space that contains T/n_w tickets. Thus, all of the results presented for lottery scheduling can also be applied to each winner in a multi-winner lottery.

The key feature of a multi-winner lottery is its ability to provide some deterministic guarantees, depending on both the distribution of tickets to clients and the value of n_w. Deterministic guarantees are based on the observation that if there is only a single client in a particular region, then that client will win the region's lottery with probability 1. Thus, each client c_i is deterministically guaranteed of receiving at least $\lfloor n_w \frac{t_i}{T} \rfloor$ quanta per superquantum. The throughput accuracy and response-time variability for client c_i depend on the fraction of its allocations that is performed deterministically. Let $d_i = \lfloor n_w \frac{t_i}{T} \rfloor$ denote the number of quanta that are deterministically allocated to client c_i per superquantum. Let $s_i = \frac{t_i}{T} - d_i$ denote the fractional number of quanta that are stochastically allocated to client c_i per superquantum.

If $s_i = 0$, the multi-winner lottery always allocates the precise number of quanta specified for client c_i during each superquantum, and there is no random error. In this case, the absolute error for client c_i equals zero at some point during every superquantum. The error at other points within a superquantum depends on the intra-superquantum schedule used to order winners. The worst-case response time for client c_i is bounded by $2(n_w - d_i) + 1$ quanta. This maximum will occur when all d_i quanta are allocated consecutively at the start of one superquantum, and then all d_i quanta are allocated consecutively at the end of the next superquantum.

If $s_i > 0$, then the throughput error for client c_i will also have a random component. This component has exactly the same properties described for ordinary lottery scheduling, where $p_i = s_i$, and n_a is replaced by the number of consecutive superquanta. If $d_i > 0$, then the maximum response time for client c_i is still bounded by $2(n_w - d_i) + 1$ quanta. Otherwise, the response time has the same geometric distribution as for lottery scheduling, with $p_i = s_i$, and n_a equal to the number of consecutive superquanta.

4.1.3 Stride Scheduling

Stride scheduling provides a strong deterministic guarantee that the absolute error for any pair of clients never exceeds one quantum. This guarantee results from the observation that the only source of pairwise error is due to quantization. A derivation of this bound is relatively straightforward. Let c_1 and c_2 denote two clients competing for a resource with a $t_1 : t_2$ ticket ratio. Let $s_1 = 1/t_1$ denote the stride for c_1, and p_1 denote the pass value for c_1. Similarly, let s_2 and p_2 denote the stride and pass values for c_2. The initial pass values are $p_1 = s_1$, and $p_2 = s_2$. For each allocation, the client c_i with the minimum pass value p_i is selected; if $p_1 = p_2$, then either client may be selected. The pass value p_i for the selected client is then advanced by s_i. Since p_1 is only advanced by s_1 when $p_1 \leq p_2$, and p_2 is only advanced by s_2 when $p_2 \leq p_1$, the maximum possible difference between p_1 and p_2 at any time is $\max(s_1, s_2)$.

The schedule produced by stride scheduling will consist of alternating sequences of allocations to clients c_1 and c_2. Without loss of generality, assume that $t_1 \geq t_2$. The absolute error for client c_2 will be greatest immediately following the longest possible sequence of allocations to client c_1. (Because there are only two clients, their absolute error values are identical, so this is also the maximum error for client c_1.) The maximum number of consecutive allocations to client c_1 is $\lceil \frac{s_2}{s_1} \rceil = \lceil \frac{t_1}{t_2} \rceil$. For this interval, the specified allocation for client c_2 is $\frac{t_2}{t_1+t_2} \lceil \frac{t_1}{t_2} \rceil$. Because the actual allocation to client c_2 over this interval is zero, its absolute error over the interval is equal to its specified allocation. Since $\lceil \frac{t_1}{t_2} \rceil \leq \frac{t_1+t_2}{t_2}$, it follows directly that the maximum absolute error for c_2 is bounded by $\frac{t_2}{t_1+t_2}\frac{t_1+t_2}{t_2} = 1$. Therefore, the maximum throughput error for any pair of clients is bounded by a single quantum. Similarly, for any pair of clients, the largest difference between the minimum and maximum response times for the same client is also bounded by one quantum. Thus, for a pair of clients, response-time distributions will be extremely tight.

Unfortunately, throughput error and response-time variability can be much larger when more than two clients are scheduled. Skewed ticket distributions can result in $O(n_c)$ absolute error, where n_c is the number of clients. For example, consider a very uneven ticket distribution in which $n_c - 1$ "small" clients each have a single ticket, and one "large" client has $n_c - 1$ tickets. The stride scheduling algorithm will schedule the large client first, and will allocate $n_c - 1$ quanta to it before any other client is scheduled. Since the specified allocation for the large client is only half that amount, its absolute error is $O(n_c)$. Response-time variability is also extremely high for such distributions, since there is no interleaving of the many small clients with the single large client.

4.1.4 Hierarchical Stride Scheduling

Hierarchical stride scheduling provides a tighter $O(\lg n_c)$ bound on absolute error, eliminating the worst-case $O(n_c)$ behavior that is possible under ordinary stride scheduling. Hierarchical stride scheduling can be analyzed as a series of pairwise allocations among successively smaller groups of clients at each level in the hierarchy. The error for each pairwise allocation is bounded by one quantum, and in the worst case, error can accumulate at each level. Thus, the maximum absolute error for a series of allocations is the height of the tree, which is $\lceil \lg n_c \rceil$ for a balanced binary tree.

The response-time characteristics of hierarchical stride scheduling are more difficult to analyze. For highly skewed ticket distributions, response-time variability can be dramatically lower than under ordinary stride scheduling. However, for other distributions, response-time variability can be significantly higher. The explanation for this behavior is that response-time variability can potentially increase multiplicatively at successive levels of the hierarchy.

Consider the first level of the tree-based data structure used for hierarchical stride scheduling. This level consists of the two children of the root node, N_l and N_r, each representing an aggregate group of clients. One of these two nodes is selected during every hierarchical allocation. The response-time distribution for each of these nodes will be tight, with a maximum difference of one quantum between its minimum and maximum values. For example, suppose that the range for the right node N_r is [3, 4]. Now consider the two children of this node, N_{rl} and N_{rr}. If all other nodes in the hierarchy are ignored, the response-time distributions for this isolated pair of nodes are also very tight; suppose that the range for the left node N_{rl} is [2, 3]. However, since node N_{rl} is only selected every second or third time that its parent N_r is selected, its overall response-time range expands to [2 × 3, 3 × 4] = [6, 12].

In general, the minimum (maximum) response time for a client can be as low (high) as the product of the minimum (maximum) response times computed in isolation for each of its ancestors. However, the actual spread may not be this large for some distributions. For example, the periodic behavior of a child may coincide with the periodic behavior of its parent if their periods divide evenly. Nevertheless, multiplicative increases in response-time variability are still possible for many distributions of tickets to clients.

4.2 Simulation Results

This section presents the results of quantitative experiments designed to evaluate the effectiveness of the various proportional-share mechanisms described in Chapter 3. The behavior of each mechanism is examined in both static and dynamic environments. As predicted by the basic analytical results, when compared to the randomized lottery-based mechanisms, the deterministic stride-based approaches generally provide significantly better throughput accuracy, with significantly lower response-time variability.

For example, Figure 4-1 presents the results of scheduling three clients with a 3 : 2 : 1 ticket ratio for 100 allocations. The dashed lines represent the ideal allocations for each client. It is clear from Figure 4-1(a) that lottery scheduling exhibits significant variability at this time scale, due to the algorithm's inherent use of randomization. The results for multi-winner lottery scheduling with $n_w = 4$, depicted in Figure 4-1(b), demonstrate reduced variability for the clients with large ticket shares. Figures 4-1(c) and 4-1(d) indicate that deterministic stride scheduling and hierarchical stride scheduling both produce the same precise periodic behavior: A, B, A, A, B, C.

The remainder of this section explores the behavior of these four scheduling mechanisms in more detail, under a variety of conditions. Throughput accuracy and response-time variability are used as the primary metrics for evaluating performance. Ideally, throughput error and response-time variability should both be minimized. However, these goals can conflict for many distributions of tickets to clients, resulting in different tradeoffs for the various scheduling techniques. For example, hierarchical stride scheduling generally minimizes throughput error, but may exhibit highly variable response times for some ticket distributions.

Although a large amount of data is presented, a regular structure has been imposed to facilitate comparisons. Figures that include results for multiple mechanisms generally consist of four rows of graphs in a fixed top-to-bottom order: lottery scheduling (L), multi-winner

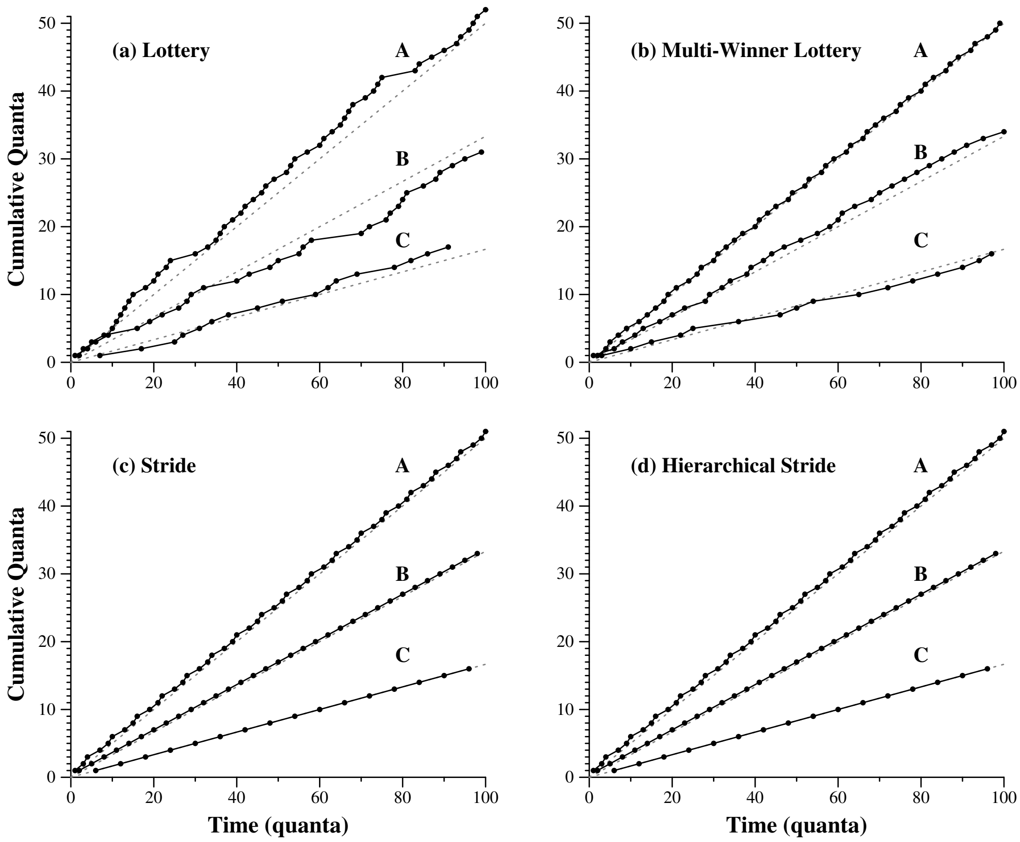

Figure 4-1: **Example Simulation Results.** Simulation results for 100 allocations involving three clients, A, B, and C, with a $3:2:1$ allocation. The dashed lines represent ideal proportional-share behavior. (a) Randomized lottery scheduler. (b) Hybrid multi-winner lottery scheduler with $n_w = 4$. (c) Deterministic stride scheduler. (d) Hierarchical stride scheduler.

lottery scheduling (M), stride scheduling (S), and hierarchical stride scheduling (H). Graphs that appear in the same column are generally associated with the same client ticket allocation, and allocations are arranged in a decreasing left-to-right order. A graph depicting results for a client with T tickets under scheduling algorithm A that appears in Figure F will be referred to as Figure $F(A, T)$.

4.2.1 Static Environment

Before considering the effects of dynamic changes, baseline behaviors are examined for a static environment. A static environment consists of a fixed set of clients, each of which has a constant ticket allocation. The first set of simulations presented involve only two clients; later simulations probe the effects of introducing additional clients.

Two Clients

Figures 4-2 and 4-3 plot the absolute error[1] and response-time distributions that result from simulating two clients under each scheduling scheme. The data depicted is representative of simulation results over a wide range of pairwise ratios. The 7 : 3 ticket ratio simulated in Figure 4-2 is typical of small ratios, and the 13 : 1 allocation simulated in Figure 4-3 is typical of large ratios.

The graphs that appear in the first column of Figures 4-2 and 4-3 plot the absolute error observed over a series of 1000 allocations. The error for the randomized lottery scheduling technique is averaged over 1000 separate runs, in order to quantify its expected behavior. The error values observed for lottery scheduling are approximately linear in $\sqrt{n_a}$, as demonstrated by Figures 4-2(L) and 4-3(L). Thus, as expected, lottery-scheduler error increases slowly with n_a, indicating that accuracy steadily improves when error is measured as a percentage of n_a. It may initially seem counterintuitive that the absolute error is considerably smaller for the 13 : 1 ratio than for the 7 : 3 ratio. The explanation for this effect is that the standard deviation for a client's actual allocation count is proportional to $\sqrt{p(1-p)}$, where p is the client's probability of winning a single lottery. For the 13 : 1 ratio, this value is roughly 0.26, while it is about 0.46 for the 7 : 3 allocation. Thus, the expected absolute error is indeed smaller for the larger ratio. However, when measured as a *percentage* of the number of allocations due to each client, the error is largest for the single-ticket client under the 13 : 1 ratio.

Three separate error curves are presented for each multi-winner lottery scheduling graph, corresponding to $n_w = 2$, 4, and 8 winners. Because multi-winner lottery scheduling has a

[1] In this case the relative and absolute errors are identical, since there are only two clients.

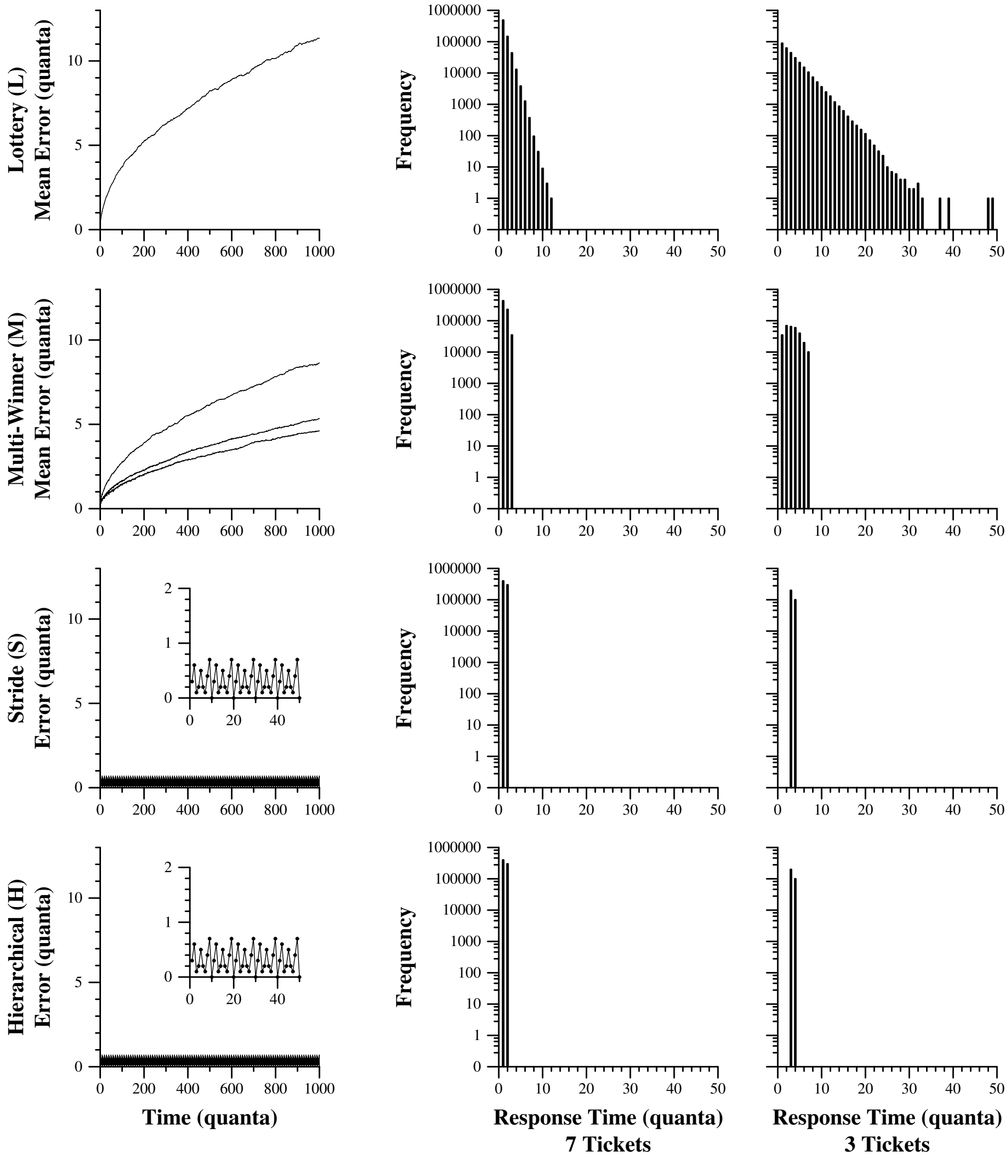

Figure 4-2: **Static Environment, 7:3 Allocation.** Simulation results for two clients with a static 7 : 3 ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. The first column graphs the absolute error measured for each mechanism over 1000 allocations. The second and third columns graph the response-time distributions for each client under each mechanism over one million allocations.

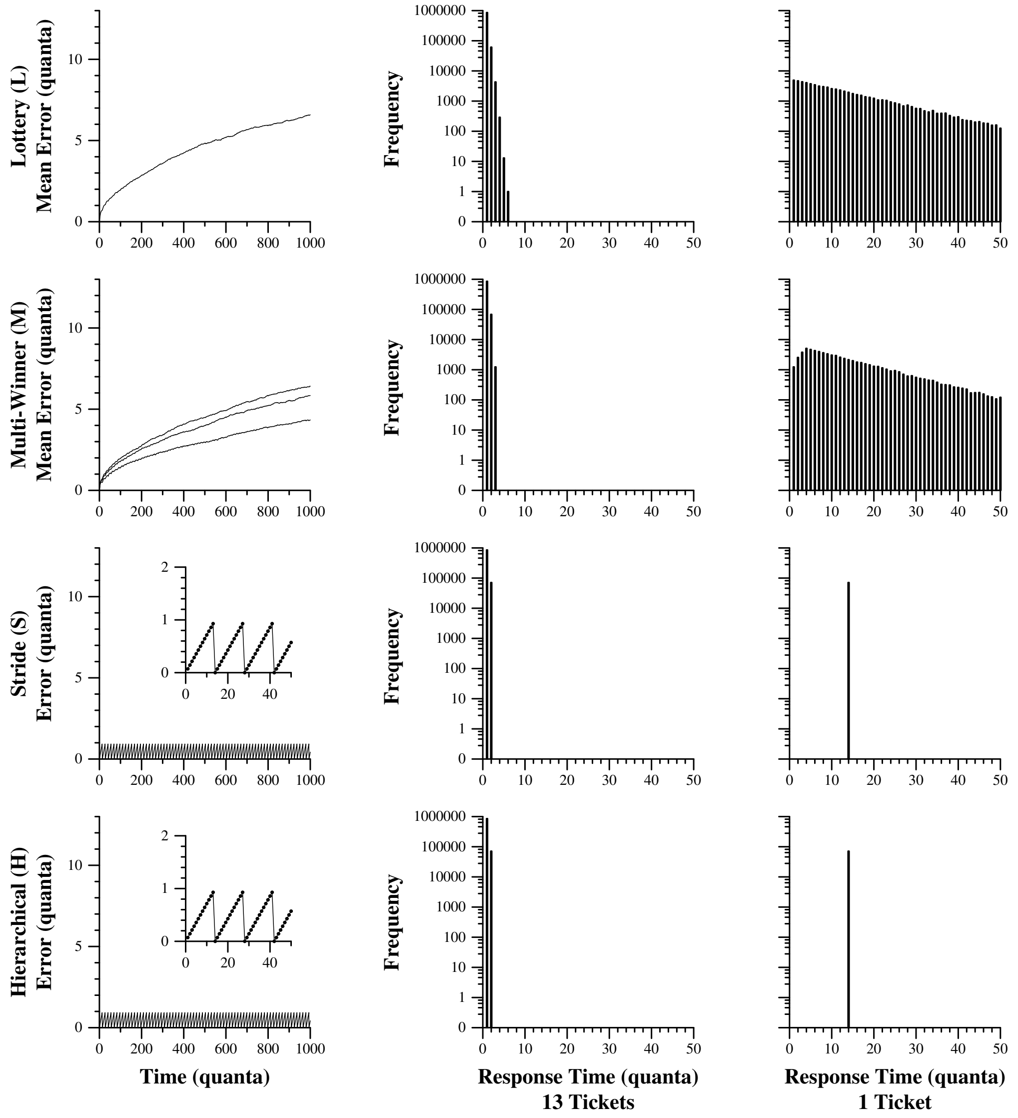

Figure 4-3: **Static Environment, 13:1 Allocation.** Simulation results for two clients with a static 13 : 1 ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. The first column graphs the absolute error measured for each mechanism over 1000 allocations. The second and third columns graph the response-time distributions for each client under each mechanism over one million allocations.

randomized component, each of these error curves is also an average over 1000 separate runs. The error curves observed for the multi-winner lotteries have the same general shape as those for a single-winner lottery. However, their absolute value is lower, often by a large amount. For the 7 : 3 ratio in Figure 4-2(M), the reductions in error are about 25%, 50%, and 60% for $n_w =$ 2, 4, and 8, respectively. For the 13 : 1 ratio in Figure 4-3(M), the corresponding reductions are approximately 3%, 10%, and 30%. As n_w increases, the deterministic component of the multi-winner lottery provides successively better approximations to the specified allocations. This decreases random error, since a smaller fraction of allocations is determined stochastically. For the 7 : 3 ratio, the deterministic approximation improves quickly, accounting for the decreasing marginal improvements as n_w increases. The larger 13 : 1 ratio exhibits the opposite behavior, since the probability that the single-ticket client will be selected during a superquantum remains below 50% until $n_w = 8$. An even larger reduction in error would result for $n_w \geq 14$.

Since stride scheduling and hierarchical stride scheduling are deterministic techniques, their absolute error curves are each plotted for a single run. As expected, the error never exceeds a single quantum, and drops to zero after each complete period – 10 quanta for the 7 : 3 ratio, and 14 quanta for the 13 : 1 allocation. This periodic behavior is clearly visible on the small insets associated with each graph. The results for both stride scheduling and hierarchical stride scheduling are identical because there are only two clients, and therefore no opportunity for aggregation under the hierarchical scheme.

The graphs that appear in the second and third columns of Figures 4-2 and 4-3 present response-time distributions for each client over one million allocations. A logarithmic scale is used for the vertical axis since response-time frequencies vary enormously across the different scheduling mechanisms. Under lottery scheduling, client response times have a geometric distribution, which appears linear on a logarithmic scale. The response-time distributions for clients with small allocations have a much longer tail than those for clients with larger allocations. This is because the standard deviation for a client's response time is $\sqrt{(1-p)}/p$, where p is the client's probability of winning a single lottery. As p approaches 1, response-time variability approaches zero; as p approaches 0, response-time variability becomes infinitely large. For the 7 : 3 ratio shown in Figure 4-2(L), the response-time distribution for the larger client drops off quickly, with a maximum of 12 quanta, while the maximum response time for the smaller client is 49 quanta. Similarly, for the 13 : 1 ratio in Figure 4-3(L), the maximum response time for the larger client is 6 quanta, but the maximum for the smaller client is off the scale at 135 quanta.

The multi-winner lottery response-time distributions are plotted for $n_w = 4$. With the 7 : 3 ratio depicted in Figure 4-2(M), this technique is extremely effective, reducing the maximum response time from 49 to 7 quanta for the smaller client. With $n_w = 4$, the deterministic approximation to the 7 : 3 ratio is very successful. However, for the 13 : 1 ratio presented in Figure 4-3(M), four winners are insufficient to provide any deterministic guarantees for the smaller client. In fact, its maximum response time actually increases to 221 quanta, although its overall distribution tightens slightly. The original single-winner distribution has a standard deviation of $\sigma = 13.50$ quanta, which is reduced to $\sigma = 12.05$ quanta with four winners.

As mentioned earlier, both stride scheduling and hierarchical stride scheduling are identical when there are only two clients. These deterministic stride-based algorithms exhibit dramatically less response-time variability than the randomized lottery-based algorithms. As expected, for both of the pairwise ratios shown in Figures 4-2(S) and 4-3(S), client response times never varied by more than a single quantum under stride scheduling. The worst-case $\sigma = 13.50$ quanta for the 13 : 1 ratio under lottery scheduling is completely eliminated under stride scheduling – all response times for the small client are *exactly* 14 quanta. The worst-case $\sigma = 2.79$ quanta for the 7 : 3 ratio under lottery scheduling is smaller by a factor of five under stride scheduling, with $\sigma = 0.47$ quanta.

Several Clients

A wider range of scheduling behavior is possible when more than two clients are considered. Figure 4-4 plots the absolute error for four clients with a 13 : 7 : 3 : 1 ticket allocation, and Figure 4-5 graphs the corresponding response-time distributions for each client. The 13 : 7 : 3 : 1 ratio was selected to allow direct comparisons with the pairwise 7 : 3 and 13 : 1 ratios used in Figures 4-2 and 4-3.

As expected, the client error curves for lottery scheduling shown in Figure 4-4(L) have the same general shape, linear in $\sqrt{n_a}$, as the pairwise error curves in Figures 4-2(L) and 4-3(L). In general, lottery scheduling is insensitive to the number of clients; each client's error is determined solely by its own relative ticket share. However, the overall number of tickets with four clients is larger than in either of the pairwise cases, so the associated reductions in relative ticket shares are reflected in the client error curves. Recall that the standard deviation for a client's actual allocation is proportional to $\sqrt{p(1-p)}$, where p is the client's probability of winning a single lottery. For the client in Figure 4-4(L,13), this value increases from approximately 0.26 to 0.50, matching the near factor-of-two increase in the client's absolute error. The error for the client in Figure 4-4(L,7) remains roughly unchanged, since its per-lottery win probability changes from 0.7 to about 0.29 ≈ (1 − 0.7). The changes in absolute error for the smaller clients also mirror the relative changes in the standard deviations for their actual allocations.