Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

John D. Regehr

May 2001

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science

John D. Regehr

Approved:

John A. Stankovic (Advisor)

Paul F. Reynolds (Chair)

Michael B. Jones

Marty A. Humphrey

Sang H. Son

Ronald D. Williams

Accepted by the School of Engineering and Applied Science:

Richard W. Miksad (Dean)

May 2001

Abstract

The CPU schedulers in general-purpose operating systems are designed to provide fast response time for interactive applications and high throughput for batch applications. The heuristics used to achieve these goals do not lend themselves to scheduling real-time applications, nor do they meet other scheduling requirements such as coordinating scheduling across several processors or machines, or enforcing isolation between applications, users, and administrative domains. Extending the scheduling subsystems of general-purpose operating systems in an ad hoc manner is time consuming and requires considerable expertise as well as source code to the operating system. Furthermore, once extended, the new scheduler may be as inflexible as the original.

The thesis of this dissertation is that extending a general-purpose operating system with a general, heterogeneous scheduling hierarchy is feasible and useful. A *hierarchy* of schedulers generalizes the role of CPU schedulers by allowing them to schedule other schedulers in addition to scheduling threads. A *general, heterogeneous* scheduling hierarchy is one that allows arbitrary (or nearly arbitrary) scheduling algorithms throughout the hierarchy. In contrast, most of the previous work on hierarchical scheduling has imposed restrictions on the schedulers used in part or all of the hierarchy.

This dissertation describes the Hierarchical Loadable Scheduler (HLS) architecture, which permits schedulers to be dynamically composed in the kernel of a general-purpose operating system. The most important characteristics of HLS, and the ones that distinguish it from previous work, are that it has demonstrated that a hierarchy of nearly arbitrary schedulers can be efficiently implemented in a general-purpose operating system, and that the behavior of a hierarchy of soft real-time schedulers can be reasoned about in order to provide guaranteed scheduling behavior to application threads. The flexibility afforded by HLS permits scheduling behavior to be tailored to meet complex requirements without encumbering users who have modest requirements with the performance and administrative costs of a complex scheduler.

Contributions of this dissertation include the following. (1) The design, prototype implementation, and performance evaluation of HLS in Windows 2000. (2) A system of *guarantees* for scheduler composition that permits reasoning about the scheduling behavior of a hierarchy of soft real-time schedulers. Guarantees assure users that application requirements can be met throughout the lifetime of the application, and also provide application developers with a model of CPU allocation to which they can program. (3) The design, implementation, and evaluation of two *augmented CPU reservation* schedulers, which provide increase scheduling predictability when low-level operating system activity *steals* time from applications.

Acknowledgments

First of all, many thanks to my advisor Jack Stankovic for being supportive of me and my work, for giving me enough rope but not too much, for pushing me in the right direction when I was wasting time or missing the point, and for speedy turnaround on many iterations of each dissertation chapter during busy times.

I'd like to thank Mike Jones for introducing me to a relevant and interesting area of research, and for being a mentor as well as a de facto second advisor.

Thanks to my committee—Marty Humphrey, Mike Jones, Paul Reynolds, Sang Son, and Ron Williams—for useful advice and constructive criticism at the proposal, at my defense, and in between. I would also like to thank Paul for being the chair of my committee and for being a patient and supportive advisor during my first few years at UVA.

Thanks to David Coppit, Gabe Ferrer, Sean McCulloch, Mike Nahas, Anand Natrajan, Alastair Reid, Glenn Wasson, and Brian White for providing useful feedback on a draft of this dissertation. Special thanks to: Sean for going above and beyond the call of duty by reading something like five chapters in two days, Dave for applying the "Kevin standard" to my claimed contributions, Brian for curbing the excesses of my (sometimes overly) parenthetical writing style, Mike and Alastair for actually reading the proofs, and Nuts for making a massive contribution.

Thanks to Sarah Creem for her love, patience, and understanding.

I would like to thank my parents for always being supportive, and for advising without pushing. Also, thanks for providing me with the 1972 World Book Encyclopedia and a Texas Instruments 99/4a—I have no idea where I'd be if I hadn't had access to these things.

Thanks to SIGBEER and the Inner Circle—you know who you are—for being, in large part, responsible for making my stay at UVA an enjoyable one and for sustaining a level of Wrongness which, although occasionally excessive, was never unwarranted.

Thanks to Jay Lepreau and the rest of the Flux group at the University of Utah for helpful conversations and friendly advice, and for giving me space in their lab while I wrote this dissertation.

Finally, I would like to thank Ginny Hilton, Brenda Lynch, and Tammy Ramsey for giving me friendly help countless times over the past six years, and for providing a much-needed level of indirection between me and the University bureaucracy.

Operating systems, in trying to meet the needs of applications, have to define what those needs are. Applications with different needs waste effort overcoming what is now a problem rather than a solution.

— *Peter Williams*, In Search of the Ideal Operating System for User Interfacing

Ideally, a general-purpose operating system that supports real-time execution should not a priori restrict the basic tenor of performance guarantees that any process is capable of obtaining.

— *Ion Stoica et al.*, A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems

A schedule defends from chaos and whim.

— *Annie Dillard*, The Writing Life

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Complementary advances in storage, processing power, network bandwidth, and data compression techniques have enabled computers to run new kinds of applications, and to run combinations of applications that were previously infeasible. For example, a modern personal computer can simultaneously decode and display a high-quality video stream, encode an audio stream in real time, and accurately recognize continuous speech; any one of these would have been impossible on an inexpensive machine just a few years ago. Also, market pressure is encouraging vendors to migrate functionality previously performed in dedicated hardware onto the main processor; this includes real-time tasks such as sound mixing [39] and modem signal processing [41]. Furthermore, home and office networks, both wired and wireless, are making personal computers into attractive storage and processing servers for resource-limited networked devices such as stereos, digital still and video cameras, and personal digital assistants. In his keynote speech at COMDEX in January 2001 [7], Intel CEO Craig Barrett said that:

> We have architected the latest generation of our microprocessor, the Pentium 4 processor, specifically for this. It was architected not to run [Microsoft] Word faster ... We did it to handle rich multimedia information. Whether it is for voice recognition, or animation or for gaming. Whether it is for showing video or capturing video or images.

Of course, powerful hardware alone is not enough—to reliably run combinations of real-time applications an operating system must effectively manage system resources such as processor time, storage bandwidth, and network bandwidth. Providing each resource to each task at an appropriate rate and granularity is no easy task; allocating at too high a rate or too fine a granularity is inefficient, and allocating at too low a rate or too coarse a granularity may reduce the value provided by applications. Scheduling is particularly difficult when the demand for one or more resources exceeds the supply—a situation that is all too common.

This dissertation focuses on the effective management of processor time, which is an important factor in overall system performance [66]. Traditional general-purpose operating systems (GPOSs) lack the flexibility required to support diverse workloads including multimedia and other soft real-time applications. They provide a single scheduling policy that is designed to support interactive and batch applications, and consequently they cannot provide application developers and users with meaningful guarantees about the level of service that applications will receive, and they cannot provide isolation between threads, applications, users, or other entities in the system. Furthermore, general-purpose operating systems give users very coarse controls for selectively allocating processor time to different applications, and they discourage potential implementors of innovative scheduling algorithms because their schedulers are interwoven with other operating system internals, and are therefore difficult to understand and modify.

1.2 The Thesis

The thesis that this dissertation supports is:

> *Extending a general-purpose operating system with general, heterogeneous hierarchical CPU scheduling is feasible and useful.*

A *hierarchy* of schedulers generalizes the role of CPU schedulers by allowing them to schedule other schedulers, as well as scheduling threads. A *general, heterogeneous* scheduling hierarchy is one that allows arbitrary (or nearly arbitrary) scheduling algorithms throughout the hierarchy. In contrast, most of the previous work on hierarchical scheduling has imposed restrictions on the schedulers used in part or all of the hierarchy. This dissertation describes the Hierarchical Loadable Scheduler (HLS) architecture, which permits schedulers to be dynamically loaded into the kernel of a general-purpose operating system.

The feasibility of general, heterogeneous hierarchical scheduling is demonstrated by (1) the design, implementation, and performance evaluation of the hierarchical scheduler infrastructure (HSI) in combination with several loadable schedulers and (2) the design of a system of *guarantees* for reasoning about the ongoing allocation of CPU time to soft real-time threads. The most important characteristics of HLS, and the ones that distinguish it from all previous work, are (1) that it has demonstrated that general, heterogeneous hierarchical scheduling can be efficiently supported by the kernel of a general-purpose operating system and that (2) the scheduling behavior of a general, heterogeneous hierarchy of soft real-time schedulers can be reasoned about in order to provide guaranteed scheduling behavior to application threads.

The usefulness of HLS is supported by showing that many deficiencies of the schedulers in general-purpose operating systems can be solved in a flexible way using a dynamically loaded hierarchy of schedulers. In particular, applications with diverse requirements can be supported by loading schedulers or combinations of schedulers that provide appropriate and, in some cases, guaranteed scheduling behavior. Guarantees provide the developers of real-time applications with a model of CPU allocation to which they can program. Guarantees also benefit end users by providing a mechanism for ensuring that the scheduling requirements of an important application will be met for the duration of the application's execution, or at least until the user wants the guarantee to end. Finally, the scheduling hierarchy supports multi-level isolation between threads, applications, and other entities.

The advantage of a flexible scheduling hierarchy over a *monolithic*, or fixed, scheduling policy is that it allows the complexity of the scheduler to be tailored to different situations. For example, a single user machine that runs a few undemanding multimedia applications can use a very simple scheduling hierarchy—the user is not forced to pay the administrative and run-time costs associated with complex scheduling behavior. Similarly, a powerful machine that must isolate the CPU usage of different users from each other while supporting multiple real-time applications can employ a much more sophisticated scheduling hierarchy—in this case users can expect their real-time applications to work (if the machine has sufficient capacity to run them) and they will not be forced to deal with unfairness issues that can result from lack of isolation. In short, we assert that there is no good "one size fits all" scheduler for general-purpose operating systems.

Additional benefits of HLS include a well-defined programming interface that can reduce the effort associated with implementing new schedulers. This is accomplished by providing loadable schedulers with the notifications about operating system events that they require in order to make scheduling decisions, while abstracting away operating-system-specific details that do not concern the scheduler. Finally, a rule-based resource manager can be used in conjunction with the scheduling hierarchy to map application threads to appropriate schedulers based on their requirements, and to enforce high-level user- and administrator-supplied policies about the allocation of processor time.

1.3 Reasons to Use and Understand Hierarchical Scheduling

The flexibility enabled by hierarchical CPU scheduling has a number of advantages:

- Variety — Multimedia applications have diverse requirements. HLS allows applications to be matched with schedulers that provide the real-time scheduling properties that they require.
- Isolation — Generalizing the role of schedulers by allowing them to schedule other schedulers allows isolation properties (such as a guaranteed share of the CPU) to be recursively applied to groups of threads, rather than applying only to single threads.
- Decomposition — HLS allows complex composite scheduling behaviors to be expressed as a collection of small, simple schedulers, providing increased flexibility compared to the "one size fits all" approach of monolithic schedulers.
- Experimentation — HLS facilitates rapid prototyping of new schedulers by allowing a new scheduler to schedule only a subset of the threads on a machine rather than taking over the job of scheduling all threads. It also permits iterations of a scheduler to be tested without rebooting the operating system.

Even in situations where it is undesirable to place an explicit scheduling hierarchy in an operating system kernel, it is useful to understand the properties of hierarchical schedulers because there are many real-world cases of *implicit hierarchical scheduling*. For example, the kernel thread scheduler in a general-purpose operating system in combination with the scheduler for a user-level thread package, the scheduler for threads in a Java virtual machine, or the kernel thread scheduler for an operating system being run in a virtual machine such as VMWare, forms a hierarchical scheduler. The kernel scheduler itself in a general-purpose operating system can be viewed as a hierarchical scheduler: a fixed-priority scheduler implemented in hardware schedules interrupts at the highest overall priority, a (usually) FIFO scheduler runs low-level kernel tasks at the middle priority, and application threads are run at the lowest priority using what is usually thought of as "the scheduler." Furthermore, when the co-resident operating system approach to real-time is used [11, 93], the kernel scheduler for a general-purpose OS is no longer the root scheduler; this privilege is taken over by a small hard-real-time OS.

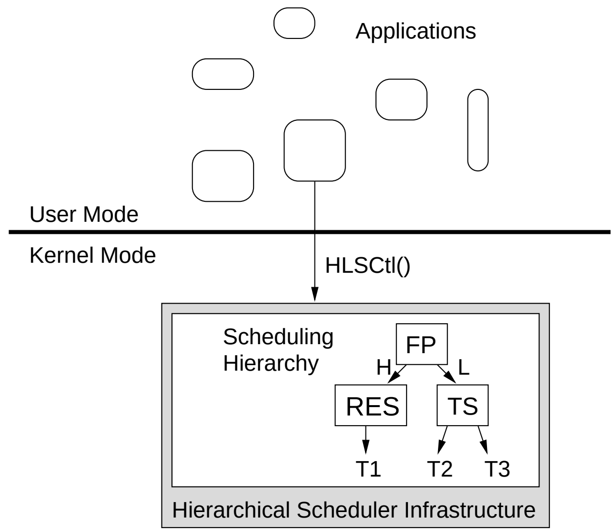

Figure 1.1: High-level overview of the HLS architecture

1.4 Overview of the HLS

Conventional time-sharing scheduling algorithms are designed with a set of tradeoffs in mind; applications running under these schedulers are forced to cope with these tradeoffs. Since general-purpose operating systems are used in diverse ways, sometimes unanticipated by the designers of the OS, it is difficult to select, in advance, the right scheduler for an operating system. A premise of HLS is that scheduling algorithms should not be chosen in advance by operating system designers. Rather, scheduling policies and combinations of scheduling policies implementing desired behaviors should be dynamically loaded into the operating system to meet the scheduling requirements of a particular usage scenario.

1.4.1 Systems Architecture

Figure 1.1 depicts the HLS architecture. The scheduling hierarchy is implemented in the kernel of a general-purpose operating system, where it is supported by the hierarchical scheduler infrastructure. There is a *root* scheduler at the top of the hierarchy that controls the allocation of all processor time. It grants the CPU to its *children*, which are below it in the hierarchy. The children may further subdivide processor time to their children, until a *leaf* of the hierarchy is reached; leaves always represent threads.

In Figure 1.1 FP, a fixed priority scheduler, is the root and RES and TS are its children. RES is a real-time scheduler and TS is a time-sharing scheduler. FP runs RES at a higher priority than TS—this means that any time RES is able to run, it is guaranteed to have access to a processor, allowing it to make real-time guarantees. It is not the case that TS can always get access to a processor: since it is scheduled at a lower priority than RES, it simply does not get to schedule any threads while RES is running. RES has one thread to schedule, T1, while TS is scheduling two threads, T2 and T3.

A request for a certain type of scheduling behavior is made by contacting the scheduling hierarchy using the `HLSCtl` command. Applications may make requests on their own, or a middleware *resource manager* may make requests on their behalf. If a resource manager is being used, it can also ensure that a particular request does not violate any user- or administrator-supplied rules about the allocation of CPU time.

1.4.2 Architecture over Time

It is important to understand the three distinct time scales over which the scheduling hierarchy interacts with applications and users.

Long: The longest-term decision that needs to be made to use HLS concerns the way a particular computer will be used—this will determine the initial shape of the scheduling hierarchy. For example, basic time-sharing usage of a personal computer requires only a single scheduler. On the other hand, providing load isolation between users while supporting multiple real-time applications with different kinds of requirements may necessitate a complex hierarchy. For a given initial hierarchy, variations in usage can be accommodated by adding, deleting, or changing parts of the hierarchy. For example, a user might at first require only time-sharing behavior, but later want to isolate the CPU usage of an untrusted application. This can be accomplished by creating a new scheduler that is a child of the main time-sharing scheduler to run threads belonging to the untrusted application. The scheduler composition logic described in Chapter 5 describes how hierarchical collections of schedulers can be shown to provide correct real-time behavior.

Medium: At the medium time scale the scheduling hierarchy remains constant while applications are instantiated, exit, and change their requirements. Decisions at this level are expected to last for seconds or minutes, and possibly for much longer. New applications are assigned scheduling from a *default* scheduler; they may then request other kinds of scheduling, or other kinds of scheduling may be requested on their behalf. The actual decision of whether a particular scheduler can grant a particular request for scheduling is made by the scheduler itself, using a *schedulability analysis* routine. For example, a voice recognition application could request an ongoing share of 10% of the CPU by sending a message to the appropriate scheduler. The return value of the message would indicate that either yes, the scheduler has enough spare capacity to reserve 10% of the processor, or no, it does not, in which case the user could manually reduce system load or try to run the application later.

Short: At the short time scale, the scheduling hierarchy and the set of applications and their requirements remain constant. This is the level at which individual scheduling decisions are made by schedulers in the hierarchy, on a granularity of milliseconds or tens of milliseconds. For example, at this level a scheduler that has guaranteed a real-time thread to receive at least 3 ms of CPU time during every 50 ms interval would use a timer to have the Hierarchical Scheduler Infrastructure send it a notification every 50 ms so that it could schedule the thread.

These time scales are intended to be descriptive of the ways that computers running general-purpose operating systems are often used, rather than being prescriptive, or telling users how to use HLS. For example, there is nothing stopping an application from requesting a different guarantee ten times per millisecond, and in fact, the entire scheduling hierarchy on a machine can be reconfigured quite rapidly. Still, it is expected that the *core scheduling hierarchy* on a machine will remain stable for a considerable length of time, often for at least for the length of time between reboots. This reflects the basic insight that most computers are purchased and used for some purpose, even if that purpose is a very general one like "running interactive and multimedia applications."

General-purpose operating systems with traditional time-sharing schedulers lack flexibility over long time scales since they have a single, fixed scheduling policy. They have some flexibility at medium time scales, but in most cases it is limited to changing the priorities of threads and limiting threads to run on a subset of the processors on a multiprocessor machine.

1.5 Scope of the Dissertation

Multimedia scheduling is a broad topic. In order to limit the scope of this dissertation a number of assumptions have been made. This section lists them and provides brief discussion and justification.

- *HLS is designed to run in a general-purpose operating system.* Virtually all personal computers and workstations run a GPOS. People use them to run multimedia applications, traditional interactive applications, background applications, and non-interactive server applications. GPOSs have been tailored to support non-real-time applications over the past several decades, and they support these applications well. It is unlikely that operating systems designed primarily to support real-time applications will replace GPOSs despite the increasing importance of multimedia and other soft real-time applications.

- *Only scheduling the CPU.* Resources other than the CPU such as memory, disk bandwidth, and network bandwidth can play an important role in overall application performance. However, while CPU scheduling is useful on its own (many applications do not require significant disk or network bandwidth), neither disk nor network bandwidth is useful without being accompanied by processor cycles to use the data. Furthermore, network quality of service is a separate topic and a large research area on its own.

- *Not attempting to make a GPOS into a real-time OS.* HLS defines an architecture for adding support for flexible scheduling to a general-purpose operating system. However, loading a real-time scheduler into a GPOS does not turn it into a real-time OS. In fact, general-purpose operating systems cannot, in general, be expected to reliably schedule real-time tasks that have sub-millisecond deadlines. However, ongoing improvements to GPOSs are being driven by the increasing importance of multimedia applications. For example, there are "low-latency" patches for Linux that reduce observed worst-case scheduling latency from tens or hundreds of milliseconds to less than 5 ms. Similarly, by avoiding hardware whose drivers are known to introduce scheduling delays, Windows 2000 can be used to attain reliable millisecond-granularity scheduling [38].

- *Applications are soft real-time, and have deadlines of at least millisecond granularity.* This assumption matches the requirements of most multimedia applications. They are soft real-time in the sense that missing a deadline is usually not catastrophic. Real-time requirements for multimedia generally come from users, who want to see a certain frame rate, reliably play stored sound files, etc. Due to the limited speed of human perceptual processing, these deadlines are usually in the 10–100 ms range.

- *Application resource requirements are known, or can be determined.* Real-time scheduling techniques rely on having an accurate estimate, for each task, of the amount of CPU time that is required and the time by which it must be received.

- *Hierarchical schedulers are trusted.* Schedulers, like other loadable device drivers, are assumed to be bug-free and to not contain malicious code. Trusting schedulers allows the entire hierarchy to be written in C and to execute in the kernel address space.

1.6 Contributions

The contributions of this dissertation fall into five categories.

Programming models: The *programming model* is identified as an important aspect of supporting multimedia applications in general-purpose operating systems. A taxonomy of multimedia programming models is presented: the first-order distinction made by the taxonomy is between the scheduling algorithms used to multiplex processor time on short time scales (milliseconds or tens of milliseconds) and the high-level mode-change protocols used to allocate CPU time over longer time periods (seconds, minutes, or longer).

Scheduler composition: The underlying unity of the scheduling behaviors provided by a broad class of multimedia scheduling algorithms is exploited in order to develop a novel system of formal guarantees about the way that schedulers allocate CPU time. This system is useful because: it separates abstract scheduling behaviors from the algorithms that provide them, it shows which guarantees are equivalent to which other guarantees, and it allows the scheduling behavior of a hierarchy of schedulers to be reasoned about. Also, a new result about the schedulability of a task set using a rate monotonic scheduler that is given a CPU reservation is presented. Finally, it is demonstrated that a number of complex, idiomatic scheduling behaviors that have been described in the literature can be implemented using HLS schedulers as basic components.

Hierarchical scheduler infrastructure: The design, implementation, and performance evaluation of an architecture supporting a hierarchy of schedulers in the kernel of a multiprocessor operating system is presented. The scheduler infrastructure is based on a novel extension of the virtual processor abstraction that was developed for *scheduler activations* [3]. It is also novel in that it is the first system that permits a hierarchy of generic scheduling algorithms to be dynamically loaded into the kernel of a general-purpose operating system. The scheduler infrastructure facilitates the implementation of new schedulers by providing a simplified programming model: it isolates loadable schedulers from OS complexities such as extraneous thread states and a difficult multiprocessor programming model. Operations on the scheduling hierarchy such as creating or destroying a scheduler instance, moving a thread between schedulers, and beginning or ending a CPU reservation can be performed quickly: they all take less than $40\,\mu$s on a 500 MHz Pentium III. Finally, although HLS increases the cost of a context switch slightly, we show that the performance penalty that a context switch imparts to a thread in terms of re-establishing its working set in the processor cache can easily be two orders of magnitude greater than the cost added by HLS.

Resource manager: The high-level design (but not detailed design or implementation) of a rule-based, user-level *resource manager* is presented. The novel feature of the resource manager is that it makes use of reflective information about the scheduling hierarchy, as well as information about users and applications, in order to implement high-level policies about the allocation of CPU time.

Augmented CPU reservations: *Stolen time* is defined and shown to be a potential obstacle to the predictable execution of real-time applications on general-purpose operating systems. For example, on a 500 MHz Pentium III running Windows 2000 more than 25% of a CPU reservation's time can be stolen by the network stack. On the same machine running Linux, close to 50% of a CPU reservation's time can be stolen by the IDE disk driver.

Two novel scheduling algorithms, Rez-C and Rez-FB, that provide *augmented CPU reservations* were designed, implemented, and evaluated. Augmented CPU reservations provide applications with increased predictability in the presence of stolen time. For example, a test application scheduled by Rez (which does not provide augmented reservations) must over-reserve by almost 25% to avoid missing deadlines when the network stack steals time from it. When scheduled by Rez-C or Rez-FB, 6% over-reservation is sufficient to eliminate nearly all deadline misses.

1.7 Outline of the Dissertation

Chapter 2 provides additional motivation for HLS by examining the relationship between the programming models that are provided by important classes of soft real-time schedulers, the properties of multimedia applications, and the conflicting requirements that must be satisfied by a general-purpose operating system that also supports multimedia applications. Chapter 3 describes three application scenarios that motivate support for flexible scheduling in general-purpose operating systems. Chapter 4 presents the design of the Hierarchical Scheduler Infrastructure (HSI)—the in-kernel component of HLS that supports loadable schedulers. In Chapter 5 the *guarantee* system for verifying the composability of hierarchical multimedia schedulers is presented. Chapter 6 continues the subject of scheduler composition, and includes a section showing that complex and useful scheduling behaviors can be composed using simple schedulers as components. Solutions to the scheduling challenges posed by the application scenarios in Chapter 3 are presented in Chapter 7. The design of the HLS resource manager is presented in Chapter 8. Chapter 9 describes the issues involved with implementing the HSI in the Windows 2000 kernel. Chapter 10 presents data about the performance of the prototype implementation. Chapter 11 describes and evaluates two schedulers that can increase predictability for applications executing on general-purpose operating systems in the presence of *stolen time*. Chapter 12 compares elements of the HLS architecture with related work that has appeared in the literature. Conclusions and future work are presented in Chapter 13. Finally, Appendix A contains an alphabetical list of technical terminology used in this dissertation, Appendix B describes the programming interface for loadable schedulers, and Appendix C contains source code for a loadable scheduler.

Chapter 2

A Survey of Multimedia Programming Models

This chapter presents background and motivation for the HLS architecture. It analyzes operating system support for soft real-time applications in terms of the different classes of schedulers that have been described in the literature and their associated programming models. Also, the requirements of several common types of multimedia applications are described in order to understand how applications can be matched with appropriate schedulers.

2.1 Introduction

The set of abstractions and conventions implemented by a particular system that allow soft real-time applications to meet their requirements defines a *programming model*. A multimedia scheduler must support a programming model that is useful and understandable to the people who develop applications for the system. Furthermore, the scheduler, in conjunction with applications, must meet user expectations and provide understandable behavior in the face of sets of applications that demand resources exceeding the capacity of the system.

For example, SMART [67] and Rialto [40] both offer deadline-based scheduling to applications in the form of *time constraints*. To use a time constraint, an application requests an amount of CPU time before a deadline (for example, 20 ms of processing during the next 100 ms); the scheduler then notifies the application that the constraint is either accepted or rejected.

Although both systems provide the same basic abstraction, the guarantees they offer are different. Once Rialto informs an application that a requested time constraint is feasible it guarantees that the time for that constraint will have been dedicated to the requesting application no matter what other applications might do. SMART, on the other hand, may invalidate a previously accepted time constraint part way through its execution, taking away its reserved time, if a higher-priority application requests a conflicting time constraint. So, SMART potentially provides faster response time for higher-priority applications that unexpectedly request CPU time, but Rialto supports a programming model in which a developer does not need to worry about the case in which a feasible time constraint is not actually scheduled.

In both cases the designers of the system believed they were making the right decision. How can it be that one man's features are another man's bugs? Clearly the authors have not agreed upon either the goals they were trying to achieve or the criteria by which their systems should be judged. Tradeoffs such as the one in this example are important because they affect the basic set of assumptions that programmers can make while implementing multimedia applications.

This chapter analyzes the sets of goals that multimedia schedulers might try to achieve. It creates a taxonomy of the kinds of programming models used to achieve these goals and it characterizes a number of representative systems used to run multimedia applications within this taxonomy. The top-level distinction made by this taxonomy is between high-level algorithms used to respond to changes in application requirements, and low-level algorithms used to make individual scheduling decisions. For each of these two broad classes we present a number of sub-categories, including representative algorithms and their important properties.

The use of this taxonomy is intended to enable: (1) careful comparisons to be made between existing work, (2) the identification of new parts of the design space leading to possible new solutions, and (3) a better understanding of how the needs of several types of multimedia applications are served by (or are not well-served by) the various programming models promoted by important types of multimedia schedulers.

2.2 Multimedia System Requirements

2.2.1 Context for the Requirements

A *general-purpose operating system* (GPOS) for a PC or workstation must provide fast response time for interactive applications, high throughput for batch applications, and some amount of fairness between applications. Although there is tension between these requirements, the lack of meaningful changes to the design of time-sharing schedulers in recent years would seem to indicate that they are working well enough.

The goal of a hard real-time system is similarly unambiguous: all hard deadlines must be met. The design of the system is dictated by this requirement, although it conflicts to some extent with designing a low-cost system. Despite the conflict, there appears to be a standard engineering practice for building such systems: statically determine resource requirements and then overprovision processor cycles as a hedge against unforeseen situations.

Not surprisingly, there is a large space of systems whose requirements fall between these two extremes. These are soft real-time systems: they need to support a dynamic mix of applications, some of which must perform computations at specific times. Missed deadlines are undesirable, but not catastrophic. In this chapter we are concerned with the requirements placed upon general-purpose operating systems that have been extended with soft real-time schedulers for the purpose of supporting multimedia applications.

We have attempted to identify an uncontroversial set of requirements that the "ideal" multimedia operating system would meet. We do not claim that the HLS architecture can meet all of these requirements, and in fact, it is unlikely that any single system can meet all of these requirements for all types of applications. Even so, the requirements are important because they describe the space within which multimedia systems are designed. A particular set of prioritizations among the requirements will result in a specific set of tradeoffs, and these tradeoffs will constrain the design of the user interface and the application programming model of a system.

2.2.2 List of Requirements

R1: *Meet the scheduling requirements of coexisting, independently written, possibly misbehaving soft real-time applications.*

The CPU requirements of a real-time application are often specified in terms of an *amount* and *period*, where the application must receive the amount of CPU time during each period of time. No matter how scheduling requirements are specified, the scheduler must be able to meet them without the benefit of global coordination among application developers—multimedia operating systems are *open systems* in the sense that applications are written independently.

A misbehaving application (from the point of view of the scheduler) will *overrun* by attempting to use more CPU time than was allocated to it. Schedulers that provide *load isolation* guarantee a minimum amount or proportion of CPU time to each multimedia application even if other applications overrun (by entering an infinite loop, for example).

R2: *Minimize development effort by providing abstractions and guarantees that are a good match for applications' requirements.*

An important role of the designers of soft real-time systems is to ease application developers into a world where their application gracefully shares machine resources with other applications. We propose the following test: compare the difficulty of writing an application for a given multimedia scheduler to the difficulty of writing the same application if it could assume that it is the highest priority application in the system (thus having the machine logically to itself). If the difference in costs is too high, application developers will assume that contention does not exist. Rather than using features provided by the scheduler, they will force their users to manually eliminate contention.

Getting the programming model right is very important: if a system becomes widely used, the effort expended by application developers will far outweigh the effort required to implement the system. It is therefore possible for small increases in usability to justify even large amounts of implementation complexity and effort. In other words, the programming model matters.

R3: *Provide a consistent, intuitive user interface.*

Users should be able to easily express their preferences to the system and the system should behave predictably in response to user actions. Also, it should give the user (or software operating on the user's behalf) feedback about the resource usage of existing applications and, when applicable, the likely effects of future actions.

R4: *Run a mix of applications that maximizes overall value.*

Unlike hard real-time systems, PCs and workstations cannot overprovision the CPU resource; demanding multimedia applications tend to use all available cycles. The multimedia OS should avoid conservatively rejecting applications that may be feasible. During overload, the multimedia OS should run a mix of applications that maximizes overall value. Value is a subjective measure of the utility of an application, running at a particular time, to a particular user.

2.3 Basis of a Taxonomy

The top level of our taxonomy of scheduling support for multimedia applications makes a distinction between low-level algorithms that make individual scheduling decisions, and higher-level algorithms that respond to application mode changes (when an application starts, terminates, or has a change of requirements). The second level of the taxonomy breaks these broad categories into

programming model	examples	load isolation	prior knowledge	support for varying latency requirements?
rate-monotonic and other static priority	Linux, RTLinux, Solaris, Windows 2000	isolated from lower priority	period, amount	yes
CPU reservations	Nemesis, Rialto, Spring	strong	period, amount	yes
proportional share	BVT, EEVDF, SMART	strong	share (, latency)	varies
earliest deadline first	Rialto, SMART	strong / weak	amount, deadline	yes
feedback control	FC-EDF, SWiFT	varies	metric, set point	varies
hierarchical scheduling	CPU Inheritance, SFQ, HLS	varies	varies	varies

Table 2.1: Characterization of soft real-time schedulers

classes of algorithms such as "CPU reservations" and "admission-control." We identify important features of these classes and place representative schedulers from the literature within them. The key questions that we answer about each category is:

- What information must applications provide to the system with in order to benefit from the scheduler?
- What kinds of guarantees are made to applications?
- What kinds of applications are supported well (and poorly)?
- Whose jobs does it make easier (and harder)?
- How comprehensible and usable is the resulting programming interface?
- How comprehensible and usable is the resulting user interface?

2.3.1 Steady State Allocation of CPU Time

For each scheduler, we provide a brief description, give examples of systems that implement it, and examine which of the requirements from Section 2.2 the scheduler fulfills. These characteristics are summarized in Table 2.1.

2.3.1.1 Static Priority and Rate Monotonic Scheduling

The uniprocessor real-time scheduling problem has essentially been solved by *static priority analysis* [4] when the set of applications and their periods and amounts are known in advance, and when applications can be trusted not to overrun. Static priority schedulers maintain the simple invariant that the runnable thread with highest priority is always scheduled. Static-priority scheduling is a generalization of *rate monotonic analysis* [53]. The core result of rate monotonic analysis is that if a set of independent periodic tasks is scheduled rate monotonically—with the shortest-period task having the highest priority, the second-shortest having the second-highest priority, and so on—then no task will ever miss a deadline as long as the total utilization over all tasks is less than 69%.

Popular general-purpose operating systems such as Linux and Windows 2000 extend their time-sharing schedulers to support static priority threads that have strictly higher priority than any time-sharing thread. Schedulers with this structure exhibit well-known pathologies such as unbounded priority inversion (unless synchronization primitives have been augmented to support priority inheritance) and starvation of time-sharing applications during overload [66]. Furthermore, developers are likely to overestimate the priority at which their applications should run because a poorly performing application reflects negatively on its author. This phenomenon is known as *priority inflation*.

BeOS has a clever requirement that, if followed, will not only eliminate priority inflation but also achieve an approximate rate-monotonic schedule: static priorities are explicitly tied to thread latency requirements, with higher priorities corresponding to shorter latencies. For example, programmers are encouraged to schedule threads with a 5–10 ms latency sensitivity at priority 110, and threads with a 0.5–1 ms latency sensitivity at priority 120 [60].

Although static priority schedulers are simple, efficient, and well understood, they fail to isolate applications from one another, and optimal priority assignment requires coordination among application developers. Applications can only be guaranteed to receive a certain amount of CPU time if the worst-case execution times of higher-priority applications are known, and this is generally not possible. Still, the static-priority programming model is reasonably intuitive for both users (if an application is starving, there must be overload at higher priorities) and programmers (higher priority applications run first), and it supports legacy applications.

2.3.1.2 CPU Reservations

A *CPU reservation* provides an application with load isolation and periodic execution. For example, a task could reserve 10 ms of CPU time out of every 50 ms; it would then be guaranteed to receive no less than the reserved amount per period. Every implementation of reservations requires an *admission test* to tell if a new reservation will overload the system, and an *enforcement mechanism* to prevent applications that exceed their allocations from using up time reserved for other tasks.

The original Spring kernel [81] is an example that represents one end of the reservation spectrum, i.e., it provides precise hard real-time guarantees. To achieve these hard guarantees Spring required significant amounts of a priori information and associated tools to extract that information. For example, the Spring programming language had restrictions placed on it such as capping all loops, no dynamic memory management, etc. Due to the cost of runtime support in this system, this solution is not suitable for continuous media. However, the Spring system was then extended by Kaneko et al. [42] to integrate continuous multimedia streams into this hard guarantee paradigm.

In general-purpose operating systems reservations can be implemented in a variety of ways. Nemesis [49] uses an earliest deadline first (EDF) scheduler in conjunction with an enforcement mechanism, and the portable Resource Kernel [68] uses a rate monotonic scheduler, the scheduler by Lin et al. [50] is driven by a table, and finally, Rialto [40] and Rialto/NT [37] provide reservations by using a tree-based data structure to represent time intervals.

CPU reservations satisfy the requirement of supporting coexisting, possibly misbehaving real-time applications. They eliminate the need for global coordination because application resource requirements are stated in *absolute* units (time) rather than *relative* units like priority or share. However, reservation-based schedulers must be told applications' periods and amounts. The period is easier to determine: the characteristics of a periodic thread, such as its data rates, buffer sizes, and latency requirements typically dictate its period; likewise, applications often implicitly make it available to the operating system by using a timer to awaken each thread every time its period begins. The amount of CPU time that an application requires can be difficult to predict, as it is both platform and data dependent. For some applications a good estimate of future amount can be obtained by averaging previous amounts; other applications such as the notoriously variable MPEG video decoder inherently show wide fluctuations in amount [8, 16]. Underestimates of amounts will sometimes prevent application requirements from being met, and overestimates will result in needless rejection of multimedia applications. Furthermore, determining CPU requirements through measurement begs the question of how to tell when a program is behaving normally and when it is overrunning.

Because reservations provide applications with fairly hard performance guarantees—how hard depends on the particular scheduler and on the characteristics of the operating system—they are best suited for scheduling applications that lose much of their value when their CPU requirements are not met. Reservations can be used to support legacy multimedia applications if the period and amount can be determined from outside the applications and applied to them without requiring modifications.

Chu and Nahrstedt [16] describe *CPU service classes*, an abstraction for giving statistical CPU reservations to highly variable applications. They give the application a base reservation that is assumed to meet its processing requirements in the common case. An *overrun partition*, a separate reservation, is shared by all periodic variable processing time (PVRT) applications. When any application in this class overruns its normal reservation, it is assigned CPU time from the overrun partition. The assumption is that since each PVRT application is overrunning a small fraction of the time, demand for the overrun partition will not often collide. In this dissertation reservations based on this idea will be called *probabilistic CPU reservations* in order to fit in with our reservation naming scheme.

2.3.1.3 Proportional Share

Proportional share schedulers are quantum-based weighted round-robin schedulers that guarantee that an application with N shares will be given at least N/T of the processor time, on average, where T is the total number of shares over all applications. This means that the absolute fraction of the CPU allocated to each application decreases as the total number of shares increases, unless the system recomputes shares each time a new application is admitted. Quantum size is chosen to provide a good balance between allocation error and system overhead.

Other than Lottery scheduling [90], which is a randomized algorithm, all proportional share algorithms appear to be based on a *virtual clock*—a per-application counter that the scheduler increments in proportion to the amount of real time the application executes and in inverse proportion to the application's share. At each reschedule, the scheduler dispatches the runnable application with the lowest virtual clock value. The differences between proportional share schedulers come from the details of how virtual times are calculated.

Some proportional share schedulers decouple an application's share from its latency requirement without actually providing any guarantee. SMART [67] is in this category: it supports a mixed programming model in which applications receiving proportional share scheduling can meet real-time requirements using the deadline-based *time constraint* abstraction. BVT [20] associates a *warp* value with each application; non-zero warp values allow a thread to build up credit while blocked, increasing the chances that it will be scheduled when it wakes up.[1] Applications with high

[1]Nemesis provides a *latency hint* that is similar to warp: it brings the effective deadline of an unblocking thread

warp values will, on average, be dispatched more quickly by BVT, but no bound on scheduling latency has been shown. The *hybrid lottery scheduler* described by Petrou et al. [71] automatically provides improved response time for interactive applications—this is an important specialization for using proportional share schedulers in real situations.

Other proportional share algorithms bound the allocation error of threads that they schedule—this a necessary condition if they are to provide a real-time guarantee. For example, both earliest eligible deadline first (EEVDF) [85] and start-time fair queuing (SFQ) [28] bound allocation error. This, in combination with admission control, allows a proportional share scheduler to provide functionality indistinguishable from a CPU reservation (we will return to this notion and formalize it in Chapter 5). Unfortunately, the period of a reservation provided by a PS scheduler is determined by the scheduler (by its allocation error bound and its quantum size) rather than by the application. So, in general, PS schedulers are best suited to scheduling applications that have latency requirements considerably longer than the scheduling quantum, or that do not require precise real-time scheduling.

2.3.1.4 Earliest Deadline First

EDF is an attractive scheduling discipline because it is optimal in the sense that if there exists any algorithm that can schedule a set of tasks without missing any deadlines, then EDF can also schedule the tasks without missing any deadlines. Soft real-time OSs primarily use EDF to keep track of deadline urgency inside the scheduler; only a few systems have exposed deadline-based scheduling abstractions to application programmers. Rialto and SMART couple deadlines with an admission test (because EDF does not work well during overload) and call the resulting abstraction a *time constraint*. The open environment for real-time applications [18] and PShED [52] provide applications with a *uniformly slower processor* (USP) abstraction that ensures they will receive a given share of the processor bandwidth over any time interval specified by the application.

Time constraints present a difficult programming model because they require fine-grained effort: the application programmer must decide which pieces of code to execute within the context of a time constraint in addition to providing the deadline and an estimate of the required processing time. Applications must also be prepared to skip part of their processing if the admission test fails. However, in Rialto, requests for time constraints are guaranteed to succeed for applications that have CPU reservations as long as the total amount of time reserved by time constraints does not exceed the rate and granularity of the reservations. Once a time constraint is accepted, Rialto guarantees the application that it will receive the required CPU time. SMART will sometimes deliver an upcall to applications informing them that a deadline previously thought to be feasible has become infeasible.

The programming model provided by uniformly slower processors is similar to the programming model supported by Rialto reservations and time constraints: each application is granted a fraction of the processor, but must dynamically notify the scheduler of its deadlines in order to meet them. However, Rialto allows time constraints to reserve extra CPU time if there is slack in the schedule, while USPs restrict the total bandwidth that is allocated to each application. Also, while Rialto did not present conditions for guaranteeing the schedulability of applications, a uniformly slower processor guarantees that any application that can be scheduled by a processor of

closer, making it more likely to be scheduled.

speed s can also be scheduled by that scheduler if it is given a USP with rate s/f on a processor of speed f. For example, assume that a task set is schedulable using an EDF scheduler on a 25 MHz processor. Then, it will also be schedulable on a 250 MHz processor if the EDF scheduler is given a uniformly slower processor with rate 0.1 (because $25/250 = 0.1$).

The proximity of a deadline alerts an EDF scheduler to the *urgency* of a task; SMART decouples urgency from *importance*, which is assumed to correlate with the value of the task to the user. SMART only considers a subset of the most important tasks when performing EDF scheduling. It gives no guarantee to applications—if an important task with a close deadline appears, other tasks with deadlines that were previously thought to be feasible will not be allocated processor time. Rialto, on the other hand, always allocates the promised processor time to an application once a deadline has been granted; this means the scheduler is not free to allocate this time to a more important task that arrives after a deadline was granted, but it also means that the Rialto programmer need not worry about the case in which a feasible deadline is not actually scheduled. By giving a stronger guarantee to the application Rialto provides an easier programming model, but SMART potentially provides faster response times to important tasks.

2.3.1.5 Feedback-Based Scheduling

Multimedia OSs need to work in situations where total load is difficult to predict and execution times of individual applications vary considerably. To address these problems new approaches based on feedback control have been developed. Feedback control concepts can be applied at admission control and/or as the scheduling algorithm itself.

In the FC-EDF work [56] a feedback controller is used to dynamically adjust CPU utilization in such a manner as to meet a specific set point stated as a deadline miss percentage. FC-EDF is not designed to prevent individual applications from missing their deadlines; rather, it aims for high utilization and low overall deadline miss ratio.

SWiFT [83] uses a feedback mechanism to estimate the amount of CPU time to reserve for applications that are structured as pipelines. The scheduler monitors the status of buffer queues between stages of the pipeline; it attempts to keep queues half full by adjusting the amount of processor time that each stage receives.

Both SWiFT and FC-EDF have the advantage of not requiring estimates of the amount of processing time that applications will need. Both systems require periodic monitoring of the metric that the feedback controller acts on. In general, feedback-based schedulers provide no guarantees to individual applications. Rather, they are designed to achieve high system utilization with few deadline misses over all applications.

2.3.1.6 Hierarchical Scheduling

Hierarchical (or multi-level) schedulers generalize the traditional role of schedulers (i.e., scheduling threads or processes) by allowing them to allocate CPU time to other schedulers. The *root* scheduler gives CPU time to a scheduler below it in the hierarchy and so on until a leaf of the scheduling tree—a thread—is reached.

The implementation of a hierarchical scheduler does not need to have a tree structure; for example, the Linux and Windows 2000 schedulers are conceptually hierarchical (the fixed-priority root scheduler always runs the static-priority scheduler if it has any ready threads and the time-sharing scheduler otherwise) but they both have a flat implementation (an array of run queues).

The scheduling hierarchy may either be fixed at system build time or dynamically constructed at run time. *CPU inheritance scheduling* [24] probably represents an endpoint on the static vs. dynamic axis: it allows arbitrary user-level threads to act as schedulers by *donating* the CPU to other threads.

Hierarchical scheduling has two important properties. First, it permits multiple programming models to be supported simultaneously, potentially enabling support for applications with diverse requirements. Second, it allows properties that schedulers usually provide to threads to be recursively applied to groups of threads. For example, a fair-share scheduler at the root of the scheduling hierarchy on a multi-user machine with a time-sharing scheduler below it for each user provides load isolation between users that is independent of the number of runnable threads each user has. A single-level time-sharing or fair-share scheduler does not do this.

Resource containers [5] and hierarchical start-time fair queuing (SFQ) [27] provide flexible isolation using hierarchical versions of proportional share schedulers. Deng et al. [18] describe a two-level scheduling hierarchy for Windows NT that has an EDF scheduler at the root of the hierarchy and an appropriate scheduler (rate-monotonic, EDF, etc.) for each real-time application. Furthermore, they developed a schedulability test that takes locally and globally synchronizing applications into account (although it relies on non-preemptive critical sections).

2.3.2 System Behavior During Mode Changes

We characterize system behavior during application mode changes by looking at the various kinds of guarantees that the operating system gives applications. The guarantee is an important part of the programming model since it determines what assumptions the programmer can make about the allocation of processor time that an application will receive.

When the OS gives an application a guarantee, it is restricting its future decision making in proportion to the strength of the guarantee. Seen in this light, it is understandable that many systems give applications weak or nonexistent guarantees—there is an inherent tradeoff between providing guarantees and dynamically optimizing value by allocating cycles on the fly in response to unexpected demand.

2.3.2.1 Best Effort

Best effort systems make no guarantees to applications. Rather than rejecting an application during overload, a best effort system reduces the processor time available to other applications to "make room" for the new one. This works well when application performance degrades gracefully.

Although "best effort" often has a negative connotation, it does not need to imply poor service. Rather, a best-effort system avoids the possibility of needlessly rejecting feasible applications by placing the burden of avoiding overload on the user. The computer and user form a feedback loop, where the user manually reduces system load after observing that applications are performing poorly.

We propose two requirements that applications must meet for "feedback through the user" to work. First, applications must degrade gracefully. Second, application performance must not be hidden from the user, who has to be able to notice degraded performance in order to do something about it. An application that fails both of these criteria is the software controlling a CD burner: it does not degrade gracefully since even a single buffer underrun will ruin a disc, and the user has no way to notice that the burner is running out of buffers supplied by the application.

2.3.2.2 Admission Control

A system that implements *admission control* keeps track of some metric of system load, and rejects new applications when load is above a threshold. For systems implementing reservations, system load could be the sum of the processor utilizations of existing reservations. The threshold for EDF-based schedulers is 100%; rate-monotonic systems can either use the worst-case bound of 69% [53] or perform the exact characterization, which has a higher run-time cost but is usually able to achieve higher utilization [48].

Because it can be used to prevent overload, admission control allows a multimedia system to meet the requirements of all admitted applications. It provides a simple programming model: applications are guaranteed to receive the amount of resources that they require until they terminate or are terminated (assuming that CPU requirements can be accurately estimated at the time a program first requests real-time service). Admission control also makes the system designer's job easy: all that is required is a load metric and a threshold.

Admission control does not serve the user well in the sense that there is no reason to believe that the most recently started application is the one that should be rejected. However, when a valuable application is denied admission the user can manually decrease the load on the system and then attempt to restart the application. Obviously, this feedback loop can fail when the admission controller rejects a job not directly initiated by the user (for example, recording a television show to disk while the user is not at home).

2.3.2.3 Resource Management: System Support for Renegotiation

Best effort and admission control are simple heuristics for achieving high overall value in situations where the user can take corrective action when the heuristic is not performing well. *Resource management* techniques attempt to achieve high overall value with little or no user intervention. They do this by stipulating that guarantees made to applications may be renegotiated to reflect changing conditions. Renegotiation is initiated when the resource manager calculates that there is a way to allocate CPU time that is different from current allocations that would provide higher value to the user. To perform this calculation, the system must have, for each application, some representation of the relationship between the resources granted to the application and the application's perceived value to the user. The Dynamic QoS Resource Manager [12, 13] and the QoS broker [65] both fall into this category. Oparah [70] describes a resource management system that extends Nemesis; it has the interesting feature that the user can assign positive or negative feedback to decisions made by the resource manager. This is designed to bring the resource manager's actions more closely into line with user preferences over time.

At the extreme end of resource management is *value maximization*, where the system understands the exact relationship between service, value, and time for each application and after any mode change chooses settings for each application that maximizes overall value [55].

2.3.2.4 Adaptation: Application Support for Renegotiation

Adaptive applications support different modes of operation along one or more dimensions. For example, a video player may support several resolutions, frame-rates, and compression methods. Each mode has a set of resource requirements and offers some value to the user. The promise of adaptive applications is that a resource manager will be able to select modes for the running set of applications that provide higher overall value than would have been possible if each application had to be either accepted at its full service rate or rejected outright.

Abdelzaher et al. [1] show that high overall value can be obtained by adapting both the period and execution time of tasks in an aircraft control simulation. The *imprecise computation model* [30] permits fine-grained adaptation by dividing computations into mandatory and optional parts, where the optional part adds value, if performed.

Assuming that an application already supports different modes, adaptation complicates the application programming model only slightly, by requiring the application to provide the system with a list of supported modes and to change modes asynchronously in response to requests from the system. Adaptive systems also require a more careful specification of what guarantees are being given to applications. For example, is an application asked if it can tolerate degraded service, is it told that it must, or does it simply receive less processor time without being notified? Is renegotiation assumed to be infrequent, or might it happen often?

Adaptation does not appear to make the user's job, the programmer's job, or the system designer's job any easier. Rather, it permits the system to provide more value to the user. A possible drawback of adapting applications is that users will not appreciate the resulting artifacts, such as windows changing size and soundtracks flipping back and forth between stereo and mono. Clearly there is a cost associated with each user-visible adaptation; successful systems will need to take this into account.

2.3.3 Practical Considerations

Programming models encompass more than high-level abstractions and APIs: any feature (or misfeature) of an operating system that the programmer must understand in order to write effective programs becomes part of the programming model. In this section we explore a few examples of this.

Can applications that block expect to meet their deadlines? Analysis of blocking and synchronization is expected for hard real-time systems; soft real-time programs are usually assumed to not block for long enough to miss their deadlines. Applications that block on calls to servers can only expect the server to complete work on their behalf in a timely [59] way if the operating system propagates the client's scheduling properties to the server, and if the server internally schedules requests accordingly. Non-threaded servers that do not perform priority queuing of requests can cause priority inversions that delay applications unnecessarily; Ingram [31] describes modifications to the X server that make it more responsive to high priority requests during periods of heavy load.

Does dispatch latency meet application requirements? Dispatch latency is the time between when a thread is scheduled and when it actually runs. It can be caused by the scheduling algorithm or by other factors; for example, in a GPOS a variety of events such as interrupt handling and network protocol processing can delay thread scheduling. Non-preemptive operating systems exacerbate the problem: a high priority thread that wakes up while the kernel is in the middle of a long system call on the behalf of another thread will not be scheduled until the system call completes. Properly configured Windows 2000 [17] and Linux [69] machines have observed worst-case dispatch latencies[2] below 10 ms—this meets the latency requirements of virtually all multimedia

[2]Based on dispatch latency measurements while the system is heavily loaded. This is not a true worst-case analysis but it indicates that the systems can perform well in practice.

type	examples	period	amount	degrades gracefully?	latency sensitivity
stored audio	MP3, AAC	around 100 ms	1%–10%	no	low
stored video	MPEG-2, AVI	33 ms	large	yes	low
distributed audio	Internet telephone	bursty	1%–10%	no	high
distributed video	video conferencing	bursty	large	yes	high
real-time audio	software synthesizer	1–20 ms	varies	no	very high
RT simulation	virtual reality, Quake	up to refresh period	usually 100%	yes	high
RT hardware	soft modem, USB speakers	3–20 ms	up to 50%	no	very high

Table 2.2: Characterization of soft real-time applications

applications. Unfortunately, their real-time performance is fragile in the sense that it can be broken by any code running in kernel mode. Device drivers are particularly problematic; rigorous testing of driver code is needed in order to reduce the likelihood of latency problems [38]. Hard real-time operating systems keep interrupt latencies very low and prohibit other kinds of unscheduled CPU time; they may have worst-case thread dispatch latencies in the tens of microseconds.

Is the same programming model available to all threads? Very low dispatch latency can be achieved using co-resident operating systems [11, 93]. This approach virtualizes the interrupt controller seen by a general-purpose operating system in order to allow a small real-time kernel to run even when the GPOS has "disabled interrupts." The GPOS runs in the idle time of the real-time kernel; the two OSs may then communicate through FIFOs that are non-blocking from the real-time side. The programming environment presented by the real-time kernel is sparse (since it cannot easily invoke services provided by the GPOS) and unforgiving (mistakes can easily hang or crash the machine). However, this is a useful approach for highly latency-sensitive applications that can be divided into real-time and non-real-time components.

2.4 Applications Characterized

The real-time requirements imposed on an operating system are driven by the applications that must be supported. This section briefly describes the main characteristics of several important categories of applications; these are summarized in Table 2.2.

There are two sources of deadlines for these applications: buffers of limited size, and the user. These sources are often interrelated. For example, speakers connected to a computer over the Universal Serial Bus (USB) have a very small amount of buffering precisely because deeper buffers would increase the time between when a sound is generated and when it is heard. A CD burner, on the other hand, could, in principle, buffer up the entire contents of the CD before beginning to write it—in this case the software controlling the CD burner would no longer be a real-time application.

2.4.1 Audio and Video

Decoding and displaying high-quality full-screen video (such as the MPEG-2 data on a DVD) in software is CPU intensive. Furthermore, the time required to decode an MPEG video stream is highly variable both on short time scales due to different types of frames and long time scales due to varying scene content [8]. Short-term variation can be smoothed over by buffering several decoded frames before displaying them.

Audio processing, on the other hand, is predictable and requires relatively little CPU time on modern processors. The end product of audio processing is a logically continuous stream of data; users are highly intolerant to holes or skips in this stream.

Audio and video applications, live or recorded, can, in principle, be adaptive. However, current applications tend to either not be adaptive, or to be manually adaptive at a coarse granularity. For example, although Winamp, a popular MP3 player, can be manually configured to reduce its CPU usage by disabling stereo sound, it has no mechanism for doing this in response to a shortage of processor cycles.

Stored audio and video: These applications are characterized by the lack of a tight end-to-end latency requirement. Reading from disk (or DVD) and decoding can be pipelined, using large buffers if necessary. The only latency-sensitive part of the process for video is switching the frame that is currently being displayed. Depending on what hardware support is available, there may be no analogous latency-sensitive operation for audio since some sound hardware can retrieve buffers from main memory as it needs them using DMA, awakening a user process only when some low-water mark is reached. Stored audio and video that arrive over the network can be treated in the same way as locally stored audio and video unless the bandwidth of the media is high enough that bursty network protocol processing becomes a problem.

Hard disk video recorders such as TiVo and Replay are dedicated devices that encode incoming television signals as MPEG-2 streams and store them to disk for later viewing. As the TiVo runs Linux and contains only a 66 MHz processor, it could easily be replaced by a PC with an MPEG-2 encoder board[3] provided that the PC were able to reserve sufficient processing (and memory and disk) resources to avoid dropping frames while encoding and decoding video. In many ways this application is a perfect motivation for soft real-time services on general-purpose operating systems because reliable real-time processing is required concurrently with, and at random times with respect to, normal PC usage. A software answering machine has similar characteristics.

Distributed live audio and video: Video conferencing and telepresence applications have a tight end-to-end latency requirement that precludes deep buffering—frames must be displayed very shortly after they are received. Network delays will cause frames to arrive in a bursty manner; this has lead to approaches such as rate-based scheduling [33].

Real-time audio: Synthesizing and mixing sounds in real-time using a PC is particularly challenging and requires end-to-end latency of not more than around 20 ms if the PC is to be used as a monitor as well as a recording device [69] (that is, if the sound is to be perceived as being simultaneous with the act of playing it). In professional situations this application is closer to hard than soft real-time because the cost of a dropped sample during a recording session may be large. Since highly reliable fine-grained (small millisecond) real-time is barely within reach of modern general-purpose OSs, this space is currently dominated by dedicated hardware solutions such as Pro Tools from Digidesign and Paris from E-MU/Ensoniq.

[3]We will not see high-quality all-software hard disk video recorders for a few years because encoding MPEG-2 is about an order of magnitude harder than decoding it [25].

2.4.2 Free-Running Real-Time Simulation

The rendering loop in immersive 3D environments and games such as Doom and Quake must display frames that depend on user input with as little delay as possible in order to be convincing and avoid inducing motion sickness. Rendering loops are usually adaptive, using extra CPU cycles to provide as many frames per second as possible, up to the screen refresh rate. These loops are usually CPU-bound since application programmers tend to make virtual environments more complex whenever the hardware appears to be getting to be fast enough.

Running other applications (real-time or not) concurrently with a free-running simulation is a matter of breaking up the time not available to the simulation into chunks small enough that individual frames are not delayed enough to degrade the user experience. For example, if 33 frames/second is an acceptable frame-rate for Quake on a platform that can provide 66 frames/second, then we would expect to be able to run applications other than Quake for 15 ms out of every 30 ms; this could be achieved by giving Quake a reservation of 15 ms / 30 ms and letting other applications use the unreserved time.

2.4.3 Real-Time Hardware

The high average-case performance of modern processors and the low profit margins in the PC industry create a powerful incentive for peripheral designers to push functionality into software. For example, software modems contain a bare minimum of hardware support, performing all signal processing tasks in software. This requires code to be reliably scheduled every 3–16 ms [46]; missed deadlines reduce throughput and may cause the connection to drop.

Since the reliable scheduling of threads at granularities of 3–10 ms is barely within the capabilities of most general-purpose operating systems, there is motivation to run soft modem code in a kernel routine (bottom-half handler in Unix or DPC in Windows) or worse, in an interrupt handler. This is “worse” in the sense that because it is usually not possible to enforce a bound on the amount of processor time given to such kernel routines, they become a source of unscheduled time for other applications. In other words, running latency-sensitive code in an interrupt or other kernel context is really only a good idea when the system designers know a priori that there are no other real-time tasks. Jones and Saroiu [41] describe the process of moving software modem driver code into thread context in Windows 2000, with the goal of allowing other real-time applications to coexist with the software modem.

USB speakers move the D/A conversion from the sound card to the speakers themselves, which receive a small sound packet from the USB controller every 1 ms. The USB controller has very little buffering, and will cause an audible skip in the played sound if data is not written to hardware for more than about 20 ms.

A trend opposing the migration of functionality into software is the decreasing size and cost of embedded computers; this makes it inexpensive to perform real-time tasks on special-purpose hardware instead of on general-purpose operating systems. However, the low cost of downloading a software module (compared to acquiring a new embedded device) ensures that users will want to perform real-time tasks on PCs during the foreseeable future. Furthermore, we believe that PCs will continue to have abundant resources compared to special-purpose devices, although PCs often lack dedicated hardware that enables some kinds of tasks to be performed much more easily.

2.5 Challenges for Practical Soft Real-Time Scheduling

In Section 2.2 we presented several requirements that a good multimedia OS should fulfill; in this section we refocus those requirements into a set of research challenges.

C1: *Create user-centric systems.* Users tell the system how to provide high value—they start up a set of applications and expect them to work. Resource management systems should respect a user's preferences when tradeoffs need to be made between applications, and should seek to maximize the utility of the system as perceived by the user. User studies are needed in order to figure out how admission control and adaptation can be used in ways that are intuitive and minimally inconvenient to users.

C2: *Create usable programming models.* In addition to the usual questions about how effective, novel, and efficient a scheduler is, we believe that the systems research community should be asking:

- What assumptions does it make about application characteristics, and are these assumptions justified?
- Can application developers use the programming model that is supported by the proposed system? Is it making their job easier?
- Are applications being given meaningful guarantees by the system?

C3: *Provide scheduling support for applications with diverse requirements.* We believe that multimedia systems should support at least three types of scheduling: (1) guaranteed rate and granularity scheduling for real-time applications that do not degrade gracefully, (2) best-effort real-time scheduling for real-time applications that degrade gracefully, and (3) time-sharing support for non-real-time applications.

2.6 Conclusions

This chapter has provided a framework by which the differing goals of many of the multimedia schedulers in research and production operating systems might be compared and evaluated. It has also shown that the multimedia and other soft real-time applications have different requirements that would be difficult to meet with a single real-time scheduling policy; this motivates the flexible and diverse scheduling that HLS enables.

Chapter 3

Application Scenarios

The application scenarios in this chapter present complex but plausible combinations of applications that expose weaknesses of current scheduling techniques and motivate flexible scheduling in general-purpose operating systems. It is essential to look at mixes of real applications since the subtleties of real applications, their requirements, and the ways that they are used have a large impact on whether or not a particular method of scheduling them is workable or not. Chapter 7 shows how each of the scheduling challenges in the application scenarios can be solved using hierarchical scheduling.

3.1 A Machine Providing Virtual Web Servers

The first application scenario involves a workstation that is being used to provide several *virtual servers*. A virtual server is a portion of a physical server that appears, at some level, to be a dedicated server. For example, virtual web servers for the domains `foo.com` and `bar.org` could reside on the same physical server.

3.1.1 Workload

Most web servers are designed to provide as much throughput as possible, rather than providing bounded delay to clients. Implementing virtual servers requires hierarchical load isolation to provide guaranteed performance to each server regardless of the CPU utilization of other virtual servers running on the same machine. For example, if `foo.com` runs many CPU-intensive database queries, this should not affect the level of service offered to `bar.org`'s users. Furthermore, virtual servers may want to subdivide resources internally in order to accommodate different classes of users, or to avoid have CPU-intensive dynamic web pages interfere with the timely serving of static content.

3.1.2 Scheduling Challenges

The scheduler for a virtual server must provide hierarchical load isolation and achieve high overall throughput on a multiprocessor machine.

3.2 A Home Computer

The second application scenario is a fast personal computer that belongs to a sophisticated user.

3.2.1 Workload

We characterize each part of the workload for this machine along two axes: foreground and background, and real-time and non-real-time. The distinction between foreground and background applications is important because although a personal computer is usually used for only one foreground application at a time, it may be used for many background tasks at once. This means that if real-time applications all fall into the foreground category, then there is little or no real need for sophisticated real-time scheduling since threads in the single real-time application can be run at high priority, eliminating contention with other applications. However, as we will see, this is not the case.

The foreground, non-real-time workload consists mainly of traditional interactive applications such as a web browser, a spreadsheet, an email application, and a word processor. Usually, all of these applications will be blocked waiting for user input, but they occasionally require bursts of CPU time.

The background, non-real-time workload consists of jobs such as: printing a document, converting the contents of a music CD into a compressed digital music format, downloading files, indexing the contents of the hard drive, and serving files to other computers over a home network.

The foreground, real-time workload for a home computer includes jobs such as playing computer games and playing stored digital video from a digital video camera, from the network, or from a DVD.

Background, real-time tasks potentially include playing music, mixing application-generated sound streams together and performing associated signal processing (to adjust the relative volumes of different streams, for example) before delivering audio buffers to sound hardware, performing voice recognition to provide input for another application such as a spreadsheet or a game, recording a television show as MPEG-2 video using a television interface card, running a software modem, burning a CD-ROM, and (eventually) using computer vision software to determine where the user is looking based on the output of a camera mounted on the monitor.

3.2.2 Scheduling Challenges

To successfully support the workload on a home computer, the scheduler must meet the scheduling requirements of any one foreground application in addition to any combination of background applications that does not overload the processor (most home computers have only a single processor).

The non-real-time workload for a home computer, both foreground and background, can be scheduled by a traditional time sharing scheduler, which was specifically designed to disambiguate interactive and batch applications, preferentially scheduling interactive applications in order to keep them responsive to user input. In a hierarchical environment, the time-sharing scheduler should receive a reserved fraction of the CPU, and should not be starved for more than about 100 ms, in order to keep interactive applications, administrative applications, and the windowing system responsive to user input. As Nieh et al. [66] observe, it is particularly important that the user interface not be starved—if it is, the user will be unable to kill the application that is causing the starvation.

The requirements of the real-time workload vary. Some applications, such as the CD burner, the software modem, the music player, and the video recorder provide little or no value if their full requirements are not met. Consequently, they should receive a reserved fraction of the CPU. Other applications such as voice recognition and vision need to be scheduled often, but do not necessarily require a reserved fraction of the processor since they continue to function correctly when they receive too little CPU time (albeit with increased delay). A third class of applications such as playing stored video and playing a game may fall into either category depending on the preferences of the user—their performance degrades gracefully, but the user may not consider any degradation to be acceptable.

A minimum share of the CPU must be reserved for CPU-bound real-time applications such as games or immersive 3-D environments. In the absence of real-time scheduling, there is no good assignment of priorities that allows these applications to make progress at the desired rate while also allowing background activity to take place on the machine. For example, if a priority-based scheduler is used to run the CPU-bound task at a higher priority than background applications, the background work will be completely starved. If the CPU-bound task is at the same priority as the background work, it will not receive processor cycles in a timely way since time-sharing schedulers enforce fairness between CPU-bound applications only in a coarse-grained manner. If the CPU-bound task runs at a lower priority than background work, it will be starved until the background work completes.

A final scheduling challenge for the home machine is to isolate the CPU utilization of untrusted applications, or of applications such as a Java virtual machine that runs untrusted applications. Isolating these applications prevents them from unfairly using more than their share of the CPU, no matter how many threads they create.

3.3 A Corporate or Departmental Terminal Server

The third application scenario involves a powerful computer (for example, a 4-processor server with several network interfaces, running Linux or Windows 2000). This machine supports several dozen users running *thin clients*. Thin clients are the modern equivalent of dumb terminals—they run a dedicated operating system that sends user input (from the keyboard and mouse) to the terminal server, and updates the display in response to commands coming from the server. All applications run on the server, which means that the client need not have a disk, much memory, or much processing power. This kind of system has advantages over giving each user a full-fledged desktop computer: system administration tasks such as software upgrades and security auditing can be centralized, and hardware is used more efficiently since a powerful machine is statistically multiplexed among many users whose peak resource requirements will not often coincide.

3.3.1 Workload

The workload for the terminal server is a subset of the applications run on a home computer, but multiplied by several dozen simultaneous users. Users run interactive applications, processor-intensive background jobs with no latency requirements, and real-time tasks such as playing audio and video. Demanding real-time tasks like voice recognition may be performed, but are less likely when each user does not have a processor to herself.

3.3.2 Scheduling Challenges

The terminal server must support multiple simultaneous non-real-time foreground and background applications. Again, a standard time sharing scheduler can accomplish this. The terminal server must also support a potentially large number of concurrent real-time tasks, and it must ensure fair scheduling across different users regardless of the number of threads each user creates. Furthermore, hierarchical load isolation may be required at other levels. For example, a development group may have large aggregate processing needs due to compilation of large jobs; it may be desirable to isolate them from non-development workers.

3.4 Coping with Inflexible Scheduling

Users have managed to use general-purpose operating systems for a long time without having access to specially tailored schedulers. It is worth examining techniques that have evolved to cope with the inflexible schedulers in GPOSs, in order to understand the benefits of flexible scheduling. Methods of coping include:

1. Manually eliminating contention by running only one real-time application at a time, or by running a combination of real-time applications whose overall utilization is low enough that their CPU requirements do not conflict.

2. Generating an ad hoc rate monotonic schedule for multiple real-time applications. In an open system, generating a rate monotonic schedule for a set of real-time applications is difficult—application priorities are determined by the developers of the applications, who at best make an educated guess about the priority at which their application should run, since they cannot know either the mix of applications that will be running or the relative importances of the set of running applications to a particular user at a particular time. Rate monotonic schedules cannot be easily constructed by end users who cannot in general be expected to either (1) know which applications have the smallest periods or (2) try out all $n!$ possible priority orderings when there are n real-time applications to be run at the same time.[1] Also, if any real-time application encounters a bug or a change in requirements and increases its CPU usage, it can cause all lower-priority tasks to miss their deadlines or become unresponsive to user input. Finally, rate monotonic scheduling techniques do not work when a real-time application (such as a game) is CPU-bound.

3. Putting time sensitive code into the operating system kernel. This effectively runs the code at the highest possible priority, potentially interfering with the schedulability of user-level tasks.

4. Using dedicated hardware to perform real-time tasks, rather than performing them on a personal computer. For example, purchasing a separate CD player, DVD player, and game console instead of using a PC to perform the associated tasks.

5. Using social mechanisms to enforce fairness. For example, on a multi-user machine fairness can be enforced by sending email to people running too many jobs, or by having a system administrator kill some jobs.

[1] There could easily be more than $n!$ combinations if applications are multithreaded.

6. Using physical isolation to enforce fairness. For example, purchasing a separate PC for each user, or purchasing a dedicated physical server for each web site rather than hosting them on virtual servers.

7. Over-provisioning the processor resource as a hedge against contention. For example, purchasing an 1100 MHz processor instead of a 750 MHz processor because the faster CPU performs background tasks so quickly that they do not have a chance to noticeably interfere with playing a game or a DVD.

The common theme among all of these solutions is that, at least in some situations, they have an associated *cost*. For example, when a machine is used to run only one real-time application at a time, the cost is the lost opportunity to run other real-time applications. When extra hardware is purchased to provide physical load isolation, the cost is the price of the equipment in addition to any extra administrative costs.

Hierarchical scheduling can give a general-purpose operating system the flexibility that is required to provide scheduling behaviors and implement scheduling policies that make most or all of the workarounds listed above unnecessary.

Chapter 4

Design of the Hierarchical Scheduler Infrastructure

This chapter describes the run-time system that was developed to allow loadable schedulers to control CPU allocation in a general-purpose operating system. The most important aspect of this description is the programming model made available to loadable schedulers, which includes: when and why different notifications are delivered to schedulers, what actions schedulers can take when responding to notifications, concurrency control in the scheduler infrastructure, and the trust model for loadable schedulers.

4.1 Goals and Requirements

The hierarchical scheduler infrastructure (HSI) is a code library that resides in the kernel of a general-purpose operating system. The goals of the HSI are (1) to allow hierarchical schedulers to efficiently allocate CPU time in such a way that the CPU requirements of soft real-time applications are met and (2) to reduce the cost of developing CPU schedulers as much as possible without violating the first goal. To meet the second goal, the HSI provides a well-defined programming interface for schedulers, insulating developers from the details of a particular OS such as the kernel concurrency model, complex thread state transitions, and the multiprocessor programming model. The following criteria ensure that the goals are met:

- Schedulers are notified of all operating system events that could necessitate a scheduling decision. For example, a scheduler should be notified when its current scheduling quantum expires, and whenever a thread it is scheduling blocks or unblocks.
- Schedulers must be able to act appropriately after being notified that an event occurred. For example, schedulers should be able to relinquish control of a processor or cause a thread to be dispatched.
- Schedulers must avoid crashing the system, hanging the system, or otherwise violating the integrity of the OS.

An additional requirement, that each scheduler must be allocated a distribution of CPU time conforming to the guarantee that it was promised, is not addressed by the HSI, which is merely a mechanism for allowing schedulers to be loaded into the OS kernel. Guarantees and scheduler composition are described in Chapters 5 and 6. A prototype implementation of the HSI in Windows 2000 is described in Chapter 9.

4.2 Entities in the Scheduler Infrastructure

Schedulers are passive code libraries that may either be built into the kernel or loaded into the kernel at run time. At least one scheduler must be built into the HSI in order to boot the OS: thread scheduling is required early in the boot process, at a time when it would be difficult to get access to the filesystem where loadable schedulers are stored. The HSI keeps a list of all schedulers (both loaded and built in), indexed by scheduler name. Scheduler implementations must not contain any static data—all scheduler state must be referenced through a scheduler instance pointer.

Scheduler instances are active entities in the scheduling hierarchy. They may be identified either by name or by a scheduler instance pointer, which points to a block of per-instance data that is allocated when the instance is created.

Virtual processors (VPs) are the principal mechanism that schedulers use to communicate with each other. Each virtual processor is shared between exactly two schedulers, and represents the potential for the parent scheduler to provide a physical CPU to the child scheduler. The separation of the scheduler instance and virtual processor abstractions is an important element of multiprocessor support in HLS. Virtual processors have a *state*—at any time they are either *running*, *ready*, or *waiting*. A virtual processor is mapped to a physical CPU only when it is running.

4.3 The Loadable Scheduler Programming Model

4.3.1 Event Notification

The CPU schedulers in general-purpose and real-time operating systems are usually state machines with some auxiliary local data. Control passes to the scheduler after some event of interest happens, at which point the scheduler updates its local data, possibly makes a scheduling decision, and then relinquishes control of the CPU in order to let a thread run. The hierarchical scheduling infrastructure was designed to efficiently support this kind of scheduler. Because function calls are a cheap, structured mechanism for control transfer, loadable schedulers use them to communicate with each other and with the OS. The function calls used to implement these notifications, as well as notifications that the HSI gives to schedulers, are listed in Table 4.1.

4.3.1.1 Virtual Processor State Change Notifications

Each scheduler is *related* to one or more other schedulers in the hierarchy through shared virtual processors. In any given relationship, a scheduler is either a parent or a child, depending on whether it is above or below the other scheduler in the scheduling hierarchy. Related schedulers notify each other of changes in the states of virtual processors they share.

Two schedulers become related when the (potential) child *registers* a virtual processor with the parent. To do this the child calls the parent's `RegisterVP` function. Similarly, when a child wishes to break a relationship with a parent, it calls the parent's `UnregisterVP` function.

A newly registered VP is in the waiting state. In order to get control of a physical processor, a scheduler must call its parent's `VP_Request` function, which puts the VP into the ready state

Notification source	Notification name	Parameters
child	`RegisterVP`	`VP, Parent, Child`
	`UnregisterVP`	`VP`
	`VP_Request`	`VP`
	`VP_Release`	`VP`
	`Msg`	`VP, MSG`
parent	`VP_Grant`	`VP`
	`VP_Revoke`	`VP`
infrastructure	`Init`	`Inst`
	`Deinit`	`Inst`
	`TimerCallback`	`Inst`
	`CheckInvar`	`Inst`

Table 4.1: Loadable scheduler entry points

pending the fulfillment of the request. At some point (possibly immediately) the parent calls the child's `VP_Grant` function indicating that the request has been granted. Before granting the request the parent sets the state of the VP to running and sets its `Proc` field to the number of one of the system's physical processors; processors in an n CPU machine are numbered $0..n-1$. When a scheduler gains control over a physical processor, it may then grant one of its own pending requests. A physical processor that was granted to one of a scheduler's VPs may be revoked by the parent scheduler at any time. When a VP is in a state other than running, its `Proc` field must be set to the special value `NO_PROC`.

A conceptual view of this relationship from the point of view of a proportional share (PS) scheduler is shown in Figure 4.1. Rounded boxes represent virtual processors. Since the PS scheduler shares two virtual processors with its parent scheduler, it is able to control the allocation of at most two physical processors. Since it shares four virtual processors with its children, it is multiplexing at most two physical processors among at most four virtual processors. From left to right, the VPs that PS shares with its parent are ready and running on processor 2. From left to right, the VPs that PS shares with its children are ready, ready, running on processor 2, and waiting.

4.3.1.2 Sending Messages to Schedulers

In addition to sending and receiving notifications about virtual processor state changes, a scheduler may send a message to its parent. Messages have no intrinsic meaning—the schedulers must agree on how to interpret them. In practice, messages are most often used to set the scheduling parameters of a VP. In the case that a scheduler is not able to send the type of message that it needs to request a particular kind of scheduling behavior, the scheduler infrastructure may send messages on its behalf in response to requests made from user-level using the `HLSCtl` interface. For example, if a time-sharing scheduler were to be assigned a CPU reservation in order to ensure fast response time and reasonable throughput for the interactive applications that it schedules, the infrastructure would send the message to the reservation scheduler requesting that it provide a CPU reservation to the time-sharing scheduler.

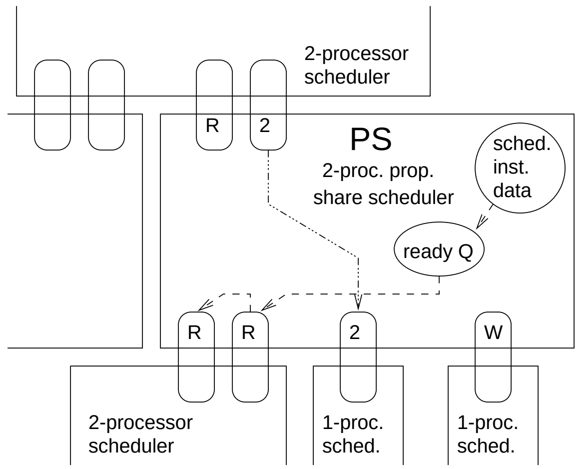

Figure 4.1: An example showing the relationship between hierarchical schedulers (rectangular boxes), virtual processors (rounded boxes), and scheduler internal data structures (ellipses). Virtual processors labeled with **R** are ready, **W** are waiting, and **2** indicates a virtual processor that is currently running on physical processor number 2.

4.3.1.3 Notifications from the Scheduler Infrastructure

The scheduler infrastructure also sends other kinds of notifications to schedulers. The `Init` function is used to set up a new instance of a scheduler, and `Deinit` is used to shut down an instance, in a manner analogous to constructors and destructors in an object-oriented system. `Init` is guaranteed to be the first notification a scheduler receives and `Deinit` the last. A scheduler will never receive the `Deinit` call while it has any children. Schedulers can set timers; when a timer expires the HSI uses the `TimerCallback` function to notify it. When debugging is enabled, the scheduler infrastructure invokes each scheduler `CheckInvar` function from time to time: this tells schedulers to run internal consistency checks in order to catch bad scheduler states as early as possible, facilitating debugging.

4.3.1.4 Design of Event Notification and Virtual Processors

The virtual processor interface is similar to, and was inspired by, the work on *scheduler activations* by Anderson et al. [3]. Scheduler activations are a mechanism for giving user-level thread schedulers running on multiprocessor machines enough information to allow them to make informed scheduling decisions. This is accomplished by notifying the user-level scheduler each time a physical processor is added to or revoked from the pool of processors available to an address space. The invariant is maintained that each user-level scheduler always knows the number of physical processors that are allocated to it, except when the final processor is revoked from it—then it learns of the revocation when a processor is next allocated to it. The applicability of scheduler activations to hierarchical scheduling is no accident: the in-kernel and user-level thread schedulers that Anderson et al. were addressing form a two-level scheduling hierarchy.

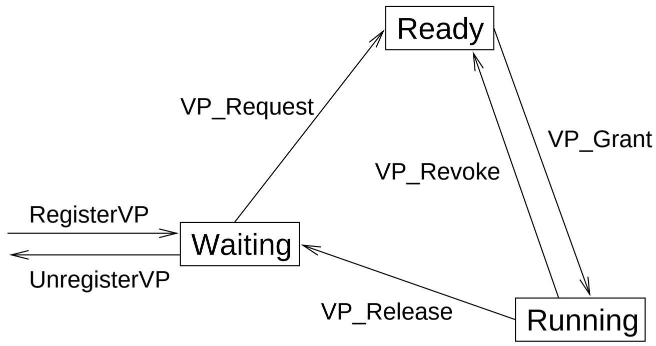

Figure 4.2: Virtual processor state transition diagram

Besides generalizing the two-level hierarchy to any number of levels, the most important differences between HLS virtual processors and scheduler activations are that (1) HLS allows the root scheduler to be chosen arbitrarily and (2) all HLS schedulers reside in the kernel. Making the root scheduler replaceable is necessary to make real-time guarantees to non-root schedulers and to applications. Putting all schedulers in the kernel allows the HSI to maintain a slightly stronger invariant than scheduler activations: each scheduler immediately learns when a processor is revoked from it, even the last one; this simplifies accurate accounting of CPU time. Also, a potential inefficiency of scheduler activations is avoided: scheduler activations increase the number of thread preemptions in the case that a processor is revoked from an address space, since a second preemption is required to tell the thread scheduler about the first one. The notification mechanism used by the HSI is virtual function call. On a 500 MHz PC running Windows 2000 a virtual function call takes about 20 ns and preempting a thread takes about 7 μs.

4.3.2 Functions Available to Schedulers

The HSI makes several functions available to loadable schedulers. They are:

- `hls_malloc` and `hls_free` — allocate and release memory.
- `HLSGetCurrentTime` — returns the current time as a 64-bit quantity in 100 ns units.
- `HLSSetTimer` — arrange to receive a timer notification at a specified time in the future.
- `HLSRegisterScheduler` and `HLSUnregisterScheduler` — inform the HSI of the presence of a new loadable scheduler, and remove a scheduler from the list of loaded schedulers.

4.3.3 Simplified States

Thread states in a general-purpose operating system may be more complicated than simply being ready, waiting, or running. For example, Windows 2000 has seven thread states: initialized, ready, waiting, transition, standby, running, and terminated. To meet the goal of simplifying the programming model for scheduler developers, HLS has only three virtual processor states, as shown in Figure 4.2. This design decision is motivated by the fact the additional states are not relevant to thread schedulers, and consequently schedulers should not be aware of them. The states of VPs that are not registered with a scheduler are undefined and are not relevant.

4.3.4 The HLS Protocol

All loadable schedulers are required to correctly implement the *HLS protocol*. Protocol violations are likely to result in unpredictable behavior and may crash the OS. The HLS protocol is derived from the requirement that, no matter what the scheduling hierarchy does internally, the end result of each notification sent to the scheduling hierarchy must be that each physical processor is running exactly one thread. This ignores transient processor activities such as switching contexts, handling interrupts, etc.—these do not concern loadable schedulers. The protocol is as follows:

- When a physical processor is granted to a hierarchical scheduler, the scheduler must immediately grant that processor to a virtual processor that has a request pending with that scheduler.
- When a physical processor is revoked from a hierarchical scheduler, it must immediately revoke that processor from the VP that it had previously granted it to.
- When a VP releases a processor, its parent must either release that processor itself, or grant it to a different child that has a pending request.
- At any time, a scheduler may request scheduling for any VP that is currently waiting.
- At any time, a scheduler may revoke a physical processor from any VP that it is currently scheduling; it must then grant the processor to a different VP or release the processor.
- VPs are, by default, waiting after being registered with another scheduler. Also, VPs may only be unregistered while they are waiting.
- All other sequences of operations on VPs are prohibited.
- The virtual processor state transition that accompanies each notification must be the one depicted in Figure 4.2.

4.3.5 Interfacing with Threads and the Dispatcher

The HLS protocol describes how hierarchical schedulers interact with each other, but not how the collection of hierarchical schedulers as a whole interacts with the rest of the operating system.

Normal hierarchical schedulers are *stackable* in the sense that they import and export the same interface. There are two special HLS schedulers that are not stackable: the *top* scheduler and *bottom* scheduler. They are not stackable since they are simply glue code for interfacing loadable schedulers with the rest of the operating system—they make no interesting scheduling decisions. The relationships between loadable schedulers and top and bottom schedulers are shown in Figure 4.3.

An instance of the bottom scheduler is automatically created for each thread in the system. Its lifetime is the same as the lifetime of the thread with which it is associated. The purpose of the bottom scheduler is to convert thread actions such as creation, blocking, unblocking, and exiting into HLS notifications. Each bottom scheduler instance creates a single VP, which it registers with a default scheduler when the thread is created. If the thread was created in a ready state, the bottom

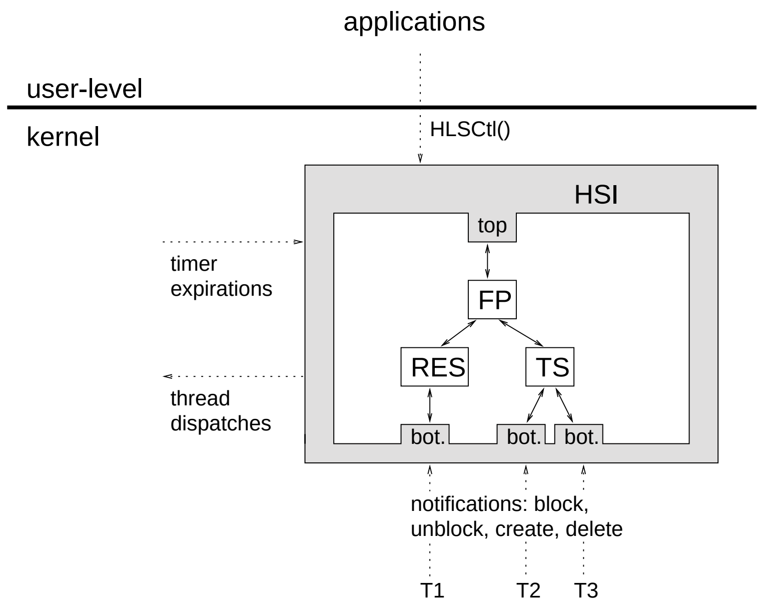

Figure 4.3: Relationship between the hierarchical scheduler infrastructure (HSI, shaded), threads (T1, T2, and T3), applications, loadable schedulers (FP, RES, and TS), and the rest of the Windows 2000 kernel

Thread event	Equivalent notification
creation	RegisterVP
deletion	UnregisterVP
unblock	VP_Request
block	VP_Release

Table 4.2: The HSI implicitly converts thread events into HLS notifications

scheduler automatically requests scheduling for its VP. When the request is granted, it dispatches the thread. When the thread blocks it implicitly releases the processor, and when the thread exits it implicitly unregisters the VP and then destroys the VP and the bottom scheduler instance. Table 4.2.

The top scheduler allows only *n* virtual processors to register with it on an *n*-processor machine. When a VP that is registered with the top scheduler makes a request, it is always immediately granted and will never be revoked. Once a VP is registered with the top scheduler, it will be granted the same physical processor every time it makes a request. When a VP that was granted a physical processor by the top scheduler releases the processor, the top scheduler runs an idle thread on that processor until it receives another request.

4.3.6 Concurrency and Preemptibility

The scheduling hierarchy is designed to execute in a serialized environment. In effect, it must be protected by a lock, meaning that even on a multiprocessor only one CPU may be executing scheduling code at a time. This design decision was made for three reasons. First, the scheduler programming model would have become considerably more difficult if scheduler authors had to worry not only about protecting shared data, but also about potential deadlocks that could occur if the hierarchy were concurrently processing several notifications. Second, since scheduler operations tend to be fast, the relative overhead of fine-grained locking inside the hierarchy would have been high. Third, all scheduling activity is serialized in Windows 2000, the operating system in which the HSI has been prototyped. Serializing the scheduling hierarchy makes it difficult to scale to large multiprocessors—this is a limitation of the current HSI design.

Hierarchical schedulers can almost always preempt user threads,[1] but schedulers themselves are not preemptible except by interrupt handlers. It is important to recognize the distinction between preemptible threads and preemptible schedulers. Schedulers execute quickly, typically taking tens of microseconds or less to make a scheduling decision, and may not be preempted by any event except a hardware interrupt. The decisions made by schedulers (for example, to run thread 17 on processor 0) are relatively long-lived, typically lasting for milliseconds or tens of milliseconds. Any scheduling decision made by a hierarchical scheduler may be overridden by a scheduler between it and the root of the scheduling hierarchy.

If more than one notification is raised while the scheduling hierarchy is processing a different notification, the order in which the new notifications are processed is undefined. For example, assume that one processor on a 4-way machine is being used by an EDF scheduler to calculate the earliest deadline of a task. While it is performing this calculation, threads on two other processors block, sending notifications to the HSI. The order in which these release notifications are sent to schedulers in the hierarchy cannot be assumed to be either (1) the first notification raised is delivered first, or (2) the notification coming from the highest-priority thread is delivered first. The order is undefined because access to the scheduling hierarchy is handled by low-level system primitives such as spinlocks and interprocessor interrupts that do not implement queuing or otherwise make ordering guarantees. Rather, the scheduling hierarchy relies on the fact that scheduler operations, spinlock acquires and releases, and interprocessor interrupts are very fast (typically taking a small number of microseconds) relative to application deadlines (typically 10 ms or more), and therefore they are unlikely to cause application deadlines to be missed. In other words, a basic simplifying assumption made by HLS is that scheduling decisions are fast compared to application deadlines.

4.3.7 Multiprocessor Schedulers

4.3.7.1 Simplifying the Multiprocessor Programming Model

One of the goals of HLS is to make it easier to develop new schedulers. Unfortunately, the low-level programming model exported by symmetric multiprocessor (SMP) machines is difficult because it is impossible for one processor to atomically induce a context switch on another processor. Rather, scheduler code running on one CPU must send an interprocessor interrupt (IPI) to the other processor, which, at some point in the future, causes scheduling code on the other processor to be called.

[1] A user thread may be non-preemptible for a short time while executing in the kernel and holding a spinlock.

In between the sending and receiving of the interprocessor interrupt, an arbitrary number of events can happen (threads blocking, unblocking, etc.), resulting in an arbitrary number of calls into the scheduler infrastructure. This implies that on a multiprocessor, a scheduler's view of which threads are running on which processors can become out of date.

To protect loadable schedulers from having to have two representations of the set of threads being scheduled (the desired and actual set of scheduled threads), the scheduler infrastructure hides from loadable schedulers the details of which thread is actually running on each processor. It presents a uniform, atomic view of the set of processors to the scheduling hierarchy, and works in the background to make the actual set of scheduled threads converge to the desired set of threads as quickly as possible.

For example, assume that a loadable scheduler running on processor 0 attempts to cause a thread running on processor 1 to be moved to processor 2. This results in two interprocessor interrupts: the first one sent to processor 1 and the second to processor 2. Before or in between the delivery of the IPIs, any number of scheduling notifications can be sent to the scheduling hierarchy, causing the loadable schedulers' view of the hierarchy to become inconsistent with the actual mapping of threads to processors. These issues are discussed in more detail in Section 9.2.3.

4.3.7.2 Processor Affinity

Schedulers that implement *processor affinity* attempt to avoid moving threads between CPUs on a multiprocessor machine because moving threads' working sets between the caches on different processors is expensive. Torrellas et al. [89] show that cache affinity can reduce overall execution time of numeric workloads by up to 10%. The HSI indirectly supports processor affinity because the top scheduler always grants the same physical processor to a given VP, allowing processor affinity to be propagated to schedulers lower in the hierarchy. As Ford and Susarla [24] observe, processor affinity will only work reliably in a scheduling hierarchy if all schedulers between a given scheduler and the root support processor affinity.

4.3.7.3 Space Sharing and Time Sharing

The HSI supports both *space sharing* and *time sharing* of multiprocessor machines. In this section, unlike the rest of the dissertation, "time sharing" refers to a way of using a machine, rather than a kind of scheduler. Space sharing means dedicating different processors (or groups of processors) on a multiprocessor to different applications. By running a different root scheduler on each processor, completely different scheduling behaviors could be given to different applications or groups of applications. Time sharing involves treating a multiprocessor as a pool of processors to be shared between applications.

Space sharing and time sharing behaviors can be mixed. For example, real-time applications can be pinned to particular processors (space sharing) while allowing interactive and batch applications to freely migrate between processors to balance load (time sharing). In general, interactive and batch applications place unpredictable demands on the system and benefit from the flexibility of time sharing behavior, while real-time tasks with predictable execution times can be space-shared effectively.

4.3.8 Treatment of Blocked Tasks

Priority inversion can prevent multimedia applications from meeting their deadlines. While it would be useful to implement a mechanism for avoiding priority inversions, HLS currently assumes that priority inversions are avoided manually. This can be accomplished by having high- and low-priority threads avoid sharing synchronization primitives, by manually raising the priority of threads before entering critical sections (that is, by manually implementing the priority ceiling protocol), or by ensuring that threads that share synchronization primitives are scheduled periodically, bounding the duration of any priority inversions.

Some priority inversions in HLS could be prevented by having a blocking thread pass its scheduling parameters to the thread that it blocks on. This is the approach taken by migrating threads [23] and CPU inheritance scheduling [24]. However, applications may block on resources such that no thread can be identified to pass scheduling parameters to. For example, a thread may block on a network device or a disk driver. Also, synchronization primitives may be used in idiomatic ways (for example, using a semaphore as a condition variable instead of a mutex) making it difficult or impossible to identify which thread will eventually wake a blocking thread. Sommer [78] presents a good survey of the issues involved in removing priority inversion in Windows NT.

4.3.9 Controlling HLS from User-Level

The `HLSCtl()` system call was added to Windows 2000 to allow user-level applications to interact with the HSI. If an implementation of the resource manager existed, it would be the only process allowed to make this call. In the current prototype implementation, any thread may do so. When the `HLSCtl` command is called, the HSI parses the arguments passed to the command and then runs the appropriate function to service the command. The operations that can be performed through `HLSCtl` are:

- Create a new instance of any scheduler at any point in the hierarchy.
- Destroy a scheduler instance.
- Begin or end a CPU reservation.
- Set the share and warp of a thread receiving proportional share scheduling.
- Move a thread between schedulers.
- Change the default scheduler, and move all threads to the scheduler that is the new default.
- Set the timer interrupt resolution to a new value.

This has proven to be a useful set of primitives for the set of loadable schedulers that has been implemented for the prototype version of HLS.

4.3.10 Trust

Loadable schedulers are *trusted*. Like other loadable kernel modules, they have access to the entire kernel address space, giving them the ability to crash the OS, read and write privileged information, control any hardware device attached to the computer, or even to completely take over the operating system.

This does not imply that untrusted users cannot load new schedulers into the kernel; rather, it means that the set of schedulers that are candidates for being loaded into the kernel must have been approved by the administrator of a particular system. Approved schedulers could be stored in a secure location or, equivalently, the HSI could store cryptographic checksums of approved schedulers in order to ensure that no non-approved scheduler is loaded into the kernel.

A loadable scheduler must correctly provide any guarantees about the allocation of CPU time that it made and also correctly implement the hierarchical scheduling protocol. A partial list of actions that a hierarchical scheduler must not take is the following:

- Interacting with the operating system through means other than the hierarchical scheduler interface (this includes reading or writing memory regions to which the scheduler was not directly given a pointer).
- Holding critical resources (such as spinlocks or the CPU) for too long. The length of time that is too long is OS dependent. For example, in Windows 2000, $25\,\mu$s is the maximum recommended spinlock hold time [61]. Since schedulers execute while a spinlock is held by the HSI, scheduling decisions should not take longer than this.
- Allocating too many system resources.
- Leaking resources.
- Causing a page fault.
- Overflowing a kernel stack.

The motivation for trusting loadable schedulers is twofold. First, trusting loadable device drivers has proven to be workable in all major general-purpose operating systems. Second, creating a language or operating system environment for implementing schedulers that meets the simultaneous goals of high performance, type safety, constrained resource usage, and guaranteed termination would have been a research project in itself.

4.3.11 Simplified Schedulers

Since a multiprocessor hierarchical scheduler may require considerable development and testing effort, it may be desirable to implement simplified schedulers in order to test new scheduling algorithms with minimal implementation effort. Section 9.4.3 presents some anecdotal evidence about the amount of time taken to develop various schedulers.

A scheduler that registers only one VP with its parent is a *uniprocessor scheduler*. These schedulers can run on multiprocessor machines, but each instance can only control the allocation of a single CPU.

A *root-only scheduler* is written under the assumption that each of its virtual processors will have exclusive access to a physical processor. Therefore, the `VP_Revoke` callback will never be called and does not need to be implemented.

Anecdotal evidence suggests that a uniprocessor, root-only proportional share scheduler can be implemented in well under a day. A few more hours of effort were required to remove the root-only assumption.

4.3.12 Supported and Unsupported Classes of Schedulers

HLS provides fully generic scheduler support in the sense that it gives schedulers all of the information about events occurring in the operating system that they need in order to make scheduling decisions. Functions are also provided for sending domain-specific information from applications to schedulers and between schedulers. Still, there are some restrictions on what schedulers can and cannot do, that may make it difficult to implement some kinds of schedulers.

Since schedulers run in a serialized environment at a higher priority than any thread, it is essential that all scheduler operations (including schedulability analysis) be completed very quickly—in tens of microseconds, preferably. Some kinds of schedulers that include complicated searches or planning-based approaches will be difficult to implement in this environment. For example, each time a new CPU reservation is requested, Rialto/NT [37] must potentially perform a complex computation to generate a new scheduling plan. When there are more than 60 existing CPU reservations, adding a new reservation may take up to about 5 ms. Since it is unacceptable to suspend all scheduling operations for 5 ms, the HLS concurrency model would have to be adjusted to support a scheduler like Rialto/NT. The most straightforward solution would be to use optimistic concurrency control to build a scheduling plan without the protection of the scheduler lock. In fact, this was the approach taken by Rialto/NT.

To use optimistic concurrency control Rialto/NT atomically (while holding the scheduler lock) copies all information that it needs to build a new scheduling plan into a temporary location and then releases the lock to allow the system to operate normally during the plan computation. Once the scheduling plan has been built, Rialto/NT reacquires the scheduler lock. The new plan must be thrown away if any event has happened during its construction that invalidates it. The difficulty of supporting compute-intensive schedulers, rather than being a property of the HLS concurrency model, is an inherent problem caused by the fact that a slow scheduler must either halt all thread operations while it runs, guaranteeing that a valid schedule will be constructed, or construct the plan concurrently with normal thread operation, which risks building an invalid plan.

Some scheduling algorithms require the ability to send asynchronous notifications to user-level applications. For example, SMART [67] supports the *time constraint* abstraction that allows an application to request a certain amount of CPU time before a deadline. If SMART informs an application that a time constraint is feasible, but the constraint later becomes infeasible (for example, due to the awakening of a higher-priority task), SMART may notify the application of the infeasible constraint, potentially allowing the application to avoid performing a useless computation. Although the HSI does not currently support sending asynchronous notification to applications, this facility could be added using abstractions commonly found in general-purpose operating systems (e.g. signals in Unix-like operating systems and APCs in Windows NT/2000).

Finally, the HSI provides no facility for network communications, which would be required to support gang scheduling, cluster co-scheduling, and other coordinated scheduling services. These

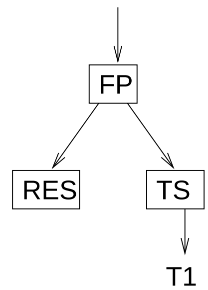

Figure 4.4: Scheduling hierarchy at the beginning of the example in Section 4.4

classes of schedulers could be implemented by adding communication facilities to HLS or by implementing schedulers that cooperate with a user-level task that shares synchronization information with other machines and passes the resulting information to the scheduler.

4.4 Scheduler Infrastructure Example

This section illustrates the operation of the hierarchical scheduling infrastructure using an example. Assume a uniprocessor PC that is running the scheduling hierarchy shown in Figure 4.4. Initially there are three scheduler instances excluding top and bottom schedulers, which are not shown in this figure. A command shell thread is blocked while waiting for user input. Since no VP has requested service, the top scheduler is running the idle thread.

Each time the user presses a key, the keyboard device driver wakes up T1, the command shell thread. T1's bottom scheduler translates the awakening into a `VP_Request` notification that it sends to its VP, which has been registered with the time-sharing scheduler (TS). The time-sharing scheduler, upon receiving this notification, must similarly request the CPU using its VP, and the fixed-priority scheduler (FP) does the same thing. When the top scheduler receives a request, it immediately grants it; FP passes the grant through to TS and then to T1's bottom scheduler, which dispatches the thread. After processing the keystroke, T1 releases the processor and the `VP_Release` notification propagates up the hierarchy until it reaches the top scheduler, which then dispatches the idle thread.

Assume that the user's keystrokes eventually form a command that launches a real-time application. A new thread T2 is created, also belonging to the default scheduler TS. When this thread's bottom scheduler requests scheduling, TS must decide which of the two threads to run; if it is the new thread, TS will revoke the processor that it had previously granted to T1 and grant it to T2.

Assume that T2 requests real-time scheduling. T2 executes the `HLSCtl` system call, which causes the scheduler infrastructure to unregister its VP from TS and to register it with the reservation scheduler (RES). The HSI also sends a message to RES requesting a CPU reservation with parameters that were passed to the kernel as part of the system call T2 made. If the reservation request is rejected, the HSI moves T2 back to TS; otherwise, T2 now has a CPU reservation.

The reservation scheduler uses a timer to receive a callback each time T2's budget needs to be replenished. At that point, if T2 is ready to run, it requests scheduling from FP. FP, by design, will always revoke the CPU from TS in response to a request from RES. TS must pass the revocation along to T1, and then FP can grant the CPU to RES, which schedules T2. To preempt T2 when its budget expires, RES arranges to wake up using a timer, revokes the CPU from T2, and then releases its VP, allowing FP to schedule TS again.

This simple example is merely intended to illustrate the interaction of schedulers, the HSI, and the operating system. More complicated motivating examples will appear in upcoming chapters. In each figure that depicts a scheduling hierarchy, schedulers will be shown in rectangular boxes and parent/child relationships (virtual processors) will be indicated by arrows.

4.5 Conclusion

This chapter has described the design of the hierarchical scheduler infrastructure and the execution environment that it provides for loadable schedulers; the programming interface for schedulers is described in more detail in Appendix B. In terms of the low-level operations described in this chapter, schedulers are simply event-driven state machines that sometimes get to control the allocation of one or more physical CPUs. However, a higher-level aspect of schedulers' behavior is critically important in a real-time system: this is the pattern of CPU time that a scheduler provides to entities that it schedules, which must conform to the *guarantee* that the scheduler made. Guarantees are the subject of the next two chapters.

Chapter 5

Composing Scheduling Policies

A thread, as a leaf node of the scheduling hierarchy, can only be scheduled when each scheduler between it and the root of the hierarchy schedules it at the same time. So, a key question is "can a given scheduling hierarchy provide threads with the guarantees that they need in order to meet their deadlines?" Clearly, some arrangements of schedulers are flawed. For example, suppose that a hierarchy includes a real-time scheduler that is scheduled using a time-sharing scheduler. Since the time-sharing scheduler makes no particular guarantees to entities that it schedules, the real-time scheduler cannot predict when it will receive CPU time, and therefore it cannot promise threads that it schedules that they will be able to meet their deadlines.

This chapter develops a system of *guarantees* about the ongoing allocation of processor time. Guarantees formalize the scheduling properties provided by a large class of multimedia schedulers, and allow these properties to be reasoned about. By matching the guarantee that one scheduler provides with the guarantee that another scheduler requires, and by using conversion rules that allow a guarantee to be rewritten as a different guarantee, complete scheduling hierarchies can be constructed with the assurance that the leaves of the hierarchy will actually receive CPU time according to the distribution they were promised.

5.1 Guarantees

5.1.1 Definition and Properties

Guarantees are the basic abstraction for reasoning about the ongoing allocation of processor cycles to real-time schedulers and threads. A guarantee is a contract between a scheduler and a virtual processor (VP) regarding the distribution of CPU time that the VP will receive for as long as the guarantee remains in force. The meaning of a particular guarantee is defined in two complementary ways:

- It is equivalent to a formal statement about the allocation of processor time that the guarantee promises. For example, a real-time thread might be guaranteed that during any time interval y units long, it will be scheduled for at least x time units.

- It is defined as the distribution of CPU time produced by a particular implementation of a scheduler.

Both aspects of a guarantee are important: the formal statement is used to reason about scheduler composition, and the correspondence with a scheduler implementation is used to provide a thread with the kind of scheduling service that it requested or was assigned. Establishing a correspondence between the two parts of the definition—proving that a particular scheduler implementation provides some scheduling behavior—is not within the scope of this dissertation.

The distinguishing characteristics of guarantees are that they describe bounds on the ongoing allocation of CPU time, and that they are independent of any particular scheduling algorithm. The primary advantages that this independence confers are:

- Guarantees abstract the behavior provided by a scheduler from the algorithm itself, permitting an application (or an entity acting on an application's behalf) to request scheduling based on its requirements, rather than requesting scheduling from a particular scheduler.

- Guarantees provide a model of CPU allocation to which developers can program. In contrast, many of the multimedia schedulers that have been proposed provide no guarantees, requiring programmers to explicitly take into account scenarios in which application threads receive amounts of CPU time other than their full requirements.

- Guarantees provide a mechanism that allows users to ensure that the requirements of important applications are always met.

- Schedulability analysis in a hierarchical scheduling system using guarantees can be performed using only local knowledge. In other words, each scheduler can determine whether or not it can provide a new guarantee based only on knowledge of the guarantee that it receives and the guarantees that it currently provides, rather than having to perform a global calculation over all schedulers in the hierarchy.

5.1.2 Composing Schedulers using Guarantees

From the point of view of the guarantee system, the purpose of the scheduling hierarchy is to convert the guarantee representing 100% of the CPU (or the set of guarantees representing 100% of multiple CPUs) into the set of guarantees required by users, applications, and other resource consumers.

Each scheduler written for HLS is characterized by one or more mappings from an *acceptable* guarantee to a set of *provided* guarantees. Any guarantee that can be converted into a guarantee that is acceptable to a scheduler is also acceptable. For example, the start-time fair queuing (SFQ) scheduler can accept a proportional share guarantee, in which case it can provide proportional share guarantees to its children. It can also accept a *proportional share with bounded error* guarantee, in which cases it can provide that guarantee to its children. The guarantee representing 100% of the CPU is acceptable to the SFQ scheduler because it can be trivially converted into either kind of proportional share guarantee.

A hierarchy of schedulers and threads *composes correctly* if and only if (1) each scheduler in the hierarchy receives a guarantee that is acceptable to it and (2) each application thread receives a guarantee that is acceptable to it. The set of guarantees that is acceptable to a scheduler is an inherent property of that scheduling algorithm. The set of guarantees that is acceptable to an application depends partially on inherent characteristics of the application, and partially on other factors, such as the application's importance.

The overall correctness of a scheduling hierarchy is established by starting at the root and working towards the children. If the scheduler at the root of a hierarchy receives the guarantee it was promised, and if each scheduler in the hierarchy correctly provides the guarantees that they have agreed to provide, then it is the case that all application threads will also receive the guarantees that they were promised.

The primary goal of the work on guarantees and scheduler composition is to ensure that each thread's guarantees can be met in a system using hierarchical scheduling. Furthermore, when translating between guarantees, as little processor time as possible should be wasted; in other words, applications' requirements should be met as efficiently as possible.

Two fundamental insights drive scheduler composition:

1. The guarantee a scheduler makes to its children can be no stronger than the guarantee that it receives; guarantees must become weaker towards the bottom of the scheduling hierarchy. The guarantee language presented in this chapter formalizes this notion.

2. Each scheduler must receive a guarantee that is semantically compatible with the guarantee that it makes.

The compositional correctness of a scheduling hierarchy can be established off-line if the hierarchy will remain fixed. Correctness can also be established online by middleware such as the resource manager.

5.1.3 Assumptions

The guarantee system for scheduler composition can ensure that a hierarchy of schedulers and threads is correct in the sense that for each application that has received a guarantee, its scheduling requirements will be met. This correctness is contingent on the following assumptions:

1. *Real-time threads are soft real-time and require ongoing guarantees.* Guarantees are assumed to be long-lived entities relative to the millisecond-granularity scheduling decisions made by the scheduling hierarchy. The assumption that threads in multimedia applications require long-lived guarantees, such as a fixed percentage of the CPU bandwidth with a bound on scheduling latency, is a valid one for many kinds of multimedia applications such as the ones surveyed in Section 2.4.

2. *Application requirements are known.* In other words, an amount of CPU time that at least meets the requirements of threads in each real-time application is known or can be calculated, either from first principles or by measurement. A procedure for determining application requirements is proposed in Section 8.5.1.

3. *Hierarchical schedulers are implemented correctly.* This means that each scheduler meets the requirements outlined in Section 4.3.10, and that when it receives the required guarantee, it correctly provides the guarantees that it has promised to entities that it schedules.

4. *Guarantees apply only while threads are runnable.* A thread's guarantee may not be met over time intervals during which it blocks. Reasons for blocking include waiting to enter a critical section, waiting for disk or network data, and waiting for the expiration of a timer. Threads in well-designed real-time applications can usually avoid blocking for so long that they miss deadlines.

5. *The operating system permits the guarantee to be provided.* In a general-purpose OS, device drivers and other kernel code executing in the context of an interrupt, a deferred procedure call, or a spinlock cannot be preempted by threads, and can therefore interfere with a scheduler's efforts to provide a guarantee. It is simply not possible to give unconditional guarantees to application threads in this kind of operating system unless the kernel and device drivers can be shown not to be non-preemptible for too long or too often. However, in practice, general-purpose operating systems like Linux and Windows 2000 are usually capable of scheduling threads in real-time multimedia applications with deadlines of several milliseconds or more. Chapter 11 discusses circumstances in which this assumption does not hold, and presents some ways to deal with the resulting problems.

5.1.4 Obtaining a Guarantee

So far, guarantees have been treated as abstract properties. In practice, however, guarantees are parameterized. For example, in the abstract a proportional share scheduler can provide a percentage of the CPU to a thread. In practice, a thread requires a specific percentage such as 10%.

To acquire this guarantee, a message is sent to the proportional share scheduler asking for a guarantee of 10% of the CPU. The scheduler uses its schedulability analysis routine to decide whether the guarantee is feasible or not. If it is, the request is accepted, otherwise it is rejected and no guarantee is provided. When a request for a guarantee is rejected, there are several courses of action that may be taken:

- System load can be reduced by terminating applications or by giving some applications a smaller guarantee.
- The requesting application can attempt to obtain a smaller guarantee.
- The guarantee given to the scheduler that rejected the request can be increased by sending a request to that scheduler's parent.

5.2 Soft Real-Time Guarantees

5.2.1 Guarantee Syntax

Guarantees are represented by identifiers of the following form:

TYPE [params]

Where TYPE represents the name of the kind of guarantee and [params] denotes a list of numeric parameters. The number of parameters is fixed for each kind of guarantee. Guarantee parameters are often in time units; for convenience, numeric parameters in this dissertation will be taken to be integers representing milliseconds unless otherwise noted.

Except in the special case of the uniformly slower processor guarantee, utilizations that appear as guarantee parameters are absolute, rather than relative. For example, if a proportional share scheduler that controls the allocation of 40% of a processor gives equal shares to two children, the guarantees that it provides have the type PS 0.2 rather than PS 0.5. Parameters in guarantees are in absolute units because this allows the meaning of each guarantee to be independent of extraneous factors such as the fraction of a processor controlled by the parent scheduler.

5.2.2 Guarantee Types

This section describes the formal properties of types of guarantees provided by multimedia schedulers.

5.2.2.1 Root Scheduler

The scheduler at the top of the hierarchy is given a guarantee of ALL, or 100% of the CPU, by the operating system. This guarantee has no parameters. Although it is tempting to parameterize this guarantee (and others) with a value indicating how much work can be done per time unit on a particular processor, Section 6.3.3.1 will present an argument that this type of parameterization is not practical in general.

5.2.2.2 CPU Reservation

The CPU reservation is a fundamental real-time abstraction that guarantees that a thread will be scheduled for a specific *amount* of time during each *period*. Reservations are a good match for threads in real-time applications whose value does not degrade gracefully if they receive less processing time than they require. A wide variety of scheduling algorithms can be used to implement CPU reservations, and many different kinds of reservations are possible. The guarantee types for CPU reservations are:

- Basic, hard: RESBH x y.
- Basic, soft: RESBS x y.
- Continuous, hard: RESCH x y.
- Continuous, soft: RESCS x y.
- Probabilistic, soft: RESPS x y z, where z is the size of the *overrun partition* that probabilistic guarantees have access to (see Section 6.2.2 for more details); probabilistic reservations are always soft.
- Non-preemptive, hard: RESNH x y (non-preemptive reservations are implicitly continuous since their CPU time is allocated as a single block that is always at the same offset within a period).
- Synchronized, hard: RESSH x y z, where z is the time that the reservation is synchronized to (synchronized reservations are implicitly continuous).

The following list describes the properties of these kinds of CPU reservations. In particular, note that every CPU reservation is either basic or continuous, and either hard or soft, and that these properties are orthogonal.

- *Basic* CPU reservations are what would be provided by an EDF or rate monotonic scheduler that limits the execution time of each thread that it schedules using a budget. Section 9.3.2 describes Rez, a basic reservation scheduler for HLS. For a reservation with amount x and period y, a basic reservation makes a guarantee to a virtual processor that there exists a time t

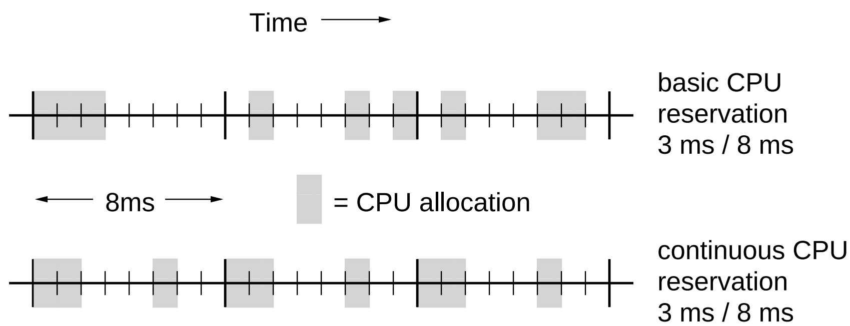

Figure 5.1: Segment of a time-line showing when the CPU is allocated to a basic and a continuous reservation

such that for every integer i the VP will receive x units of CPU time during the time interval $[t+iy, t+(i+1)y]$. In other words, the reservation scheduler divides time into period-sized intervals, during each of which it guarantees the VP to receive the reserved amount of CPU time. The value of t is chosen by the scheduler and is not made available to the application.

- *Continuous* CPU reservations (as defined by Jones et al. [40]) are those that make the following guarantee: given a reservation with amount x and period y, for any time t, the thread will be scheduled for x time units during the time interval $[t, t+y]$. A continuous CPU reservation is a stronger guarantee than a basic CPU reservation since every period-sized time interval will contain the reserved amount of CPU time, rather than only certain scheduler-chosen intervals. In other words, a continuous reservation scheduler is not free to arbitrarily rearrange CPU time within a period. Continuous reservations are provided by schedulers that utilize a (possibly dynamically computed) static schedule, such as Rialto [40] and Rialto/NT [37], where a thread with a reservation receives its CPU time at the same offset during every period. In contrast, a basic reservation scheduler such as Rez retains the freedom to schedule a task during any time interval (or combination of shorter intervals) x units long within each time interval y units long. Figure 5.1 depicts two CPU reservations, one basic and one continuous, that are guaranteed to receive 3 ms of CPU time out of every 8 ms. The continuous CPU reservation is also a basic reservation, but the basic reservation is not a continuous reservation.

- *Hard* reservations limit the CPU usage of a virtual processor to at most the reserved amount of time, as well as guaranteeing that it will be able to run at at least that rate and granularity. Hard reservations are useful for applications that cannot opportunistically take advantage of extra CPU time; for example, those that display video frames at a particular rate. They are also useful for limiting the CPU usage of applications that were written to use the full CPU bandwidth provided by a processor slower than the one on which they are currently running. For example, older CPU-bound games and voice recognition software have been run successfully on fast machines by limiting their utilization to a fraction of the full processor bandwidth.

- *Soft* reservations may receive extra CPU time on a best-effort basis. They are useful for applications that can use extra CPU time to provide added value. However, no extra time is guaranteed.

- *Probabilistic* reservations (called CPU service classes by Chu and Nahrstedt [16]), are a special case of soft CPU reservations that guarantee a thread to receive a specified minimum execution rate and granularity as well as having a chance of obtaining extra CPU time. Probabilistic reservations are designed to give applications with highly variable execution times a high probability of meeting their deadlines without actually giving them a guarantee based on their worst-case execution times.

- *Non-preemptive* reservations guarantee that the amount of reserved time will be provided to the virtual processor in a single block, at the same offset in each period. In other words, for a non-preemptive reservation with amount x and period y there exists a time t such that for every integer i, the thread will be scheduled for the time interval $[t+iy, t+iy+x]$. This guarantee is useful for threads in applications that must interact with sensitive hardware devices requiring the undivided attention of the CPU. Although any reservation provided by the main thread scheduler in a GPOS is always preemptible by hardware and software interrupts, these events are generally very short compared to time quanta for application threads.

- *Synchronized* reservations are a special case of non-preemptive reservations that start at a specific time. Synchronized reservations permit the requesting thread to specify a time t such that for every integer i, the thread will be scheduled for the time interval $[t+iy, t+iy+x]$. Synchronized reservations are useful to synchronize a reservation to a periodic external event (such as a video card refresh—this is a real, and difficult problem), or to implement schedulers that coordinate thread execution on different processors of a multiprocessor, or on different machines.

5.2.2.3 Uniformly Slower Processors

A uniformly slower processor (USP) is a special kind of CPU reservation that was described in Section 2.3.1.4. A USP guarantee has the form RESU r, where r is the fraction of the overall CPU bandwidth allocated to the USP. The granularity over which the reserved fraction of the CPU is to be received is not part of the guarantee, and must be specified dynamically. The guarantee provided by a uniformly slower processor is as follows. Given a virtual processor with guarantee RESU r, for any two consecutive deadlines d_n and d_{n+1} that the child scheduler at the other end of the VP notifies the USP scheduler of, the VP is guaranteed to receive $r(d_{n+1} - d_n)$ units of CPU time between times d_n and d_{n+1}.

5.2.2.4 Proportional Share

Proportional share (PS) schedulers are quantum-based approximations of fair schedulers. Some PS schedulers can guarantee that during any time interval of length t, a thread with a share s of the total processor bandwidth will receive at least $st - \delta$ units of processor time, where δ is an error term that depends on the particular scheduling algorithm. This guarantee is called "proportional share bounded error" and has the type PSBE s δ. For the *earliest eligible virtual deadline first* (EEVDF) scheduler [34], δ is the length of the scheduling quantum. For the *start-time fair queuing* (SFQ) scheduler [28], δ is more complicated to compute: it is a function of the quantum size, the number of threads being scheduled by the SFQ scheduler, and the share of a particular thread.

Since it is particularly well suited to hierarchical scheduling, we will return to the SFQ scheduler in Sections 5.3.1.4 and 5.3.3.

Some proportional share schedulers, such as the *lottery scheduler* [90], provide no deterministic performance bound to threads that they schedule. The guarantee given by this kind of PS scheduler is PS s, where s is the asymptotic share that the thread is guaranteed to receive over an unspecified period of time.

5.2.2.5 Time Sharing

Since time-sharing schedulers make no particular guarantee to threads they schedule, they make the NULL guarantee, indicating strictly best-effort scheduling. Time-sharing schedulers usually attempt (but do not really guarantee) to avoid completely starving threads because starvation has been shown to lead to unbounded *priority inversion* when threads are starved while holding shared resources. Priority inversion is said to occur when a low-priority thread holds a resource, preventing a high-priority thread from running.

Most multilevel feedback queue time-sharing schedulers avoid starving threads because they are actually weak proportional share schedulers. The Windows 2000 scheduler (which is not proportional share) avoids starvation by increasing a thread's priority to the maximum time-sharing priority if it is runnable but does not get to run for more than about three seconds. Both of these schedulers attempt to ensure that threads will not be starved as long as there are no high-priority "real-time" threads.

Time-sharing (TS) schedulers are designed to avoid starving threads that they schedule on a best-effort basis. However, TS schedulers will be unable to prevent starvation if the schedulers themselves are starved. It is therefore desirable to ensure that each time-sharing scheduler in a system receives a real (instead of best-effort) scheduling guarantee. This guarantee will ensure that (1) priority inversions caused by time-sharing threads being starved while holding resources do not occur, and (2) that the windowing system and associated administrative applications will remain responsive to user input regardless of the behavior of real-time applications in the system.

5.2.3 Completeness of the Set of Guarantees

The set of guarantees presented in this chapter is by no means complete. Rather, it covers an interesting and viable set of guarantees made by multimedia schedulers that have been presented in the literature and that, together, can be used to meet the scheduling needs of many kinds of multimedia applications.

An additional guarantee type based on the *time constraint* abstraction provided by Rialto or Rialto/NT could be defined—time constraints guarantee a certain amount of CPU time before an application-specified deadline. However, this guarantee would be a dead-end with respect to conversion to other kinds of guarantees using schedulers and rewrite rules. Furthermore, the one-shot constraints violate our assumption that guarantees are long-lived entities.

An interesting guarantee would be one based on Mok and Chen's *multiframe tasks* [63]. Multiframe tasks exploit patterns in applications' CPU requirements in order to avoid making reservations based on worst-case requirements. This would be useful for applications such as the MPEG decoder, which exhibit large but somewhat predictable fluctuations in requirements between frames [8].

Scheduler	Conversions
Fixed Priority	any $\mapsto$ (any, NULL$^+$)
Join	*see below*
Limit	RESBS $\mapsto$ RESBH
PS	PS $\mapsto$ PS$^+$, PSBE $\mapsto$ PSBE$^+$, RESU $\mapsto$ PSBE$^+$, RESCH $\mapsto$ PSBE$^+$
Rez	ALL $\mapsto$ RESBH$^+$, RESU $\mapsto$ RESBH$^+$
TS	NULL $\mapsto$ NULL$^+$
BSS-I, PShED	ALL $\mapsto$ RESU$^+$, RESU $\mapsto$ RESU$^+$
BVT	PS $\mapsto$ PS$^+$, RESU $\mapsto$ PS$^+$, RESCH $\mapsto$ PS$^+$
CBS	ALL $\mapsto$ RESBH$^+$, RESU $\mapsto$ RESBH$^+$
EEVDF	ALL $\mapsto$ PSBE$^+$, RESU $\mapsto$ PSBE$^+$
Linux	NULL $\mapsto$ NULL$^+$
Lottery	PS $\mapsto$ PS$^+$, RESU $\mapsto$ PS$^+$, RESCH $\mapsto$ PS$^+$
Resource Kernel	ALL $\mapsto$ (RESBS$^+$, RESBH$^+$), RESU $\mapsto$ (RESBS$^+$, RESBH$^+$)
Rialto, Rialto/NT	ALL $\mapsto$ RESCS$^+$, RESU $\mapsto$ RESCS$^+$
SFQ	PS $\mapsto$ PS$^+$, PSBE $\mapsto$ PSBE$^+$, RESU $\mapsto$ PSBE$^+$, RESCH $\mapsto$ PSBE$^+$
SFS	PS$^+$ $\mapsto$ PS$^+$, RESU$^+$ $\mapsto$ PS$^+$, RESCH$^+$ $\mapsto$ PS$^+$
SMART	NULL $\mapsto$ NULL$^+$
Spring	ALL $\mapsto$ RESBH$^+$, RESU $\mapsto$ RESBH$^+$
Stride	PS $\mapsto$ PS$^+$, RESU $\mapsto$ PS$^+$, RESCH $\mapsto$ PS$^+$
TBS	ALL $\mapsto$ RESBS$^+$, RESU $\mapsto$ RESBH$^+$
Windows 2000	NULL $\mapsto$ NULL$^+$

Table 5.1: Guarantees required and provided by common multimedia scheduling algorithms and scheduler implementations. Notation used in this table is explained in Section 5.3.1.

5.3 Converting Between Guarantees

Recall that from the point of view of guarantees, the purpose of the scheduling hierarchy is to convert the ALL guarantee(s) into the set of guarantees required by users, applications, and other resource consumers. The remainder of this chapter describes the two ways that guarantees can be converted into other guarantees. First, each scheduler in the hierarchy requires a guarantee, and provides guarantees to other schedulable entities through virtual processors. Second, *guarantee rewrite rules* can be used to convert guarantees without using a scheduler to perform the conversion.

5.3.1 Converting Guarantees Using Schedulers

Table 5.1 shows a number of schedulers and what guarantee(s) they can provide. The top six schedulers have been implemented in HLS; the remaining schedulers have been described in the literature. The table is to be interpreted as follows:

- A $\mapsto$ B$^+$ means that a scheduler can convert a guarantee with type A into multiple guarantees of type B. A is the weakest guarantee that is acceptable to the scheduler when used to provide guarantees of type B. Implicitly, any guarantee that can be converted into guarantee A using one of the conversions in Section 5.3.2 is also acceptable.

- The identifier "any" indicates a variable that may be bound to any guarantee. So, the fixed priority scheduler passes whatever guarantee it is given to its highest-priority virtual processor while providing multiple NULL guarantees to lower-priority VPs.
- Whenever a scheduler makes use of the RESU guarantee, amounts of CPU time in the guarantees that it provides must be interpreted as being in the domain of the uniformly slower processor. For example, if a reservation scheduler that receives a guarantee of RESU 0.5 provides a reservation of RESBS 10 20, the thread that receives that reservation will only have 25% of the total CPU bandwidth available to it, because $0.5(10/20) = 0.25$.

Sections 5.3.1.1 through 5.3.1.7 discuss and justify the guarantee conversions listed in Table 5.1.

5.3.1.1 Fixed Priority

A *preemptive, fixed-priority* scheduler that uses admission control to schedule only one virtual processor at each priority gives no guarantee of its own: rather, it passes whatever guarantee it receives to its highest-priority child. All other children receive the NULL guarantee.

This logic can be seen to be correct by observing that no other virtual processor can create scheduling contention for the highest-priority VP when a preemptive fixed-priority scheduler is in use. No guarantee can be given to lower-priority VPs because the one with the highest priority may be CPU-bound.

5.3.1.2 Join

Most uniprocessor schedulers register a single virtual processor with their parent, and multiplex CPU time received from that virtual processor among multiple children. The *join* scheduler performs the opposite function: it registers multiple virtual processors with its parents and schedules its child VP any time any of the parents allocates a physical processor to it. This allows the scheduling hierarchy to be generalized to a directed acyclic graph.

Join schedulers function as OR gates in the scheduling hierarchy: they schedule their child virtual processor when receiving CPU time from any parent. The synthesis of complex scheduling behaviors from collections of small, simple schedulers often requires the use of join schedulers. For example, as Section 6.2.1 illustrates, a scheduler that provides hard CPU reservations can also be used to provide soft CPU reservations using join schedulers. The purpose of a join scheduler, as in the preceding example, is usually to direct slack time in the schedule to a virtual processor that can make use of it. For this reason, a join scheduler will usually be joining a guarantee such as a CPU reservation with a NULL guarantee (since slack time in the schedule is, by definition, not guaranteed). Even so, the rest of this section will present the exact rules for the guarantees that a join scheduler can provide, which were too complicated to present in Table 5.1.

An entity scheduled by a join scheduler may pick any one of its parent guarantees to take, with the restriction that if the guarantee that it picks is a hard guarantee, it must be converted into the corresponding soft guarantee. To see that this is correct, notice that a join scheduler cannot give a virtual processor any less CPU time than the VP would have received if it were directly given any of the join scheduler's parent guarantees. However, the join scheduler may cause a virtual processor to receive additional CPU time, meaning that it cannot give a hard guarantee.

On a uniprocessor machine, guarantees of the same type (or guarantees that can be converted into the same type) may be added together. For example, if a join scheduler receives guarantees of RESBS 2 30 and RESBS 3 30, then the join scheduler can provide a guarantee of RESBS 5 30. Clearly this is more complicated when the reservations have different periods. We do not work out this case in detail since it is difficult to picture this situation coming up in practice—it would be simpler and more efficient to simply have a single scheduler give the thread the guarantee that it needs. Guarantees cannot be added on a multiprocessor machine because the join scheduler may be scheduled by multiple parents at the same time, and it can make use of at most one. To address this case, multiprocessor join schedulers must have a method for deciding from which parent to accept a physical processor. For example, a join scheduler could assign unique numbers to its parents and always release any processors granted from parents other than the highest-numbered parent that grants a processor.

5.3.1.3 Limit

Limit schedulers can be used to convert a soft guarantee into a hard one. A limit scheduler that is given a guarantee of a basic, soft CPU reservation would, like a reservation scheduler, keep track of the amount of CPU time that it has allocated to its (single) child virtual processor during each period. However, when its child is about to receive more than the guaranteed amount, it releases the processor and does not request it again until the start of the next period.

5.3.1.4 Proportional Share

The PS scheduler that was implemented for HLS implements the *start-time fair queuing* (SFQ) algorithm with a *warp* extension similar to the one in BVT [20]. When the warp of all virtual processors is zero, it behaves as an SFQ scheduler. Goyal et al. showed that an SFQ scheduler provides fair resource allocation in a hierarchical scheduling environment where it does not receive the full CPU bandwidth. This result was derived in the context of network scheduling [28] and has also been applied to CPU scheduling [27]. Therefore, the conversion PS $\mapsto$ PS$^+$ is justified. They also showed that when an SFQ scheduler is scheduled by a *fluctuation constrained* (FC) server, then the entities scheduled by the SFQ scheduler are also FC servers.

We now show that an FC server is the same as the PSBE (proportional share bounded error) guarantee. Following the notation in Lee [47, p. 127], an FC server is characterized by two parameters (s, δ). Informally, s is the average share guaranteed to a virtual processor and δ is the furthest behind the average share it may fall. Let $C(t)$ denote the instantaneous service delivered by a processor. For any times a and b with $a < b$

$$\int_a^b C(t)dt \geq \max(0, s(b-a) - \delta) \tag{5.1}$$

Therefore, the FC server constrains the deviation from the average service rate. This is precisely what the PSBE guarantee does, and consequently an FC server with parameters (s, δ) is equivalent to the guarantee PSBE s δ.

Therefore, an SFQ scheduler that is given a PSBE guarantee can also provide this guarantee to its children. The formula for calculating the parameters of the child FC servers was derived in [28]. Here we present a simplified version of that equation since the original accounts for variable-length scheduling quanta—this is unnecessary when SFQ is used to schedule the CPU, which uses a periodic clock interrupt to provide a uniform quantum size.[1]

Let q be the scheduler quantum size and T be the total number of threads being scheduled by a SFQ scheduler, with r_f being the weight (the fraction of the total number of shares) assigned to thread f. Then, if an SFQ scheduler is scheduled by an FC server with parameters (s, δ), each of the threads scheduled by the SFQ scheduler is also a FC server with parameters calculated as follows:

$$\left(sr_f,\ r_f\frac{Tq}{s}+r_f\frac{\delta}{s}+q\right) \tag{5.2}$$

The utility of this result will become clear in Section 5.3.2 where we show that it is possible to convert a proportional share bounded error guarantee into a CPU reservation and vice-versa.

Table 5.1 shows that proportional share scheduler may make use of the uniformly slower processor guarantee. Despite the fact that PS schedulers do not have any notion of deadlines, this can be accomplished by providing a uniform fraction of the CPU to the PS scheduler during each scheduling quantum.

Theorem 5.1. *Any proportional share scheduler can be scheduled correctly by (1) a continuous CPU reservation whose period is equal to the size of the scheduling quantum, or (2) a uniformly slower processor by treating the end of each scheduling quantum as a deadline.*

Informal proof. Since the time quantum of a PS scheduler determines its minimum enforceable scheduling granularity, the scheduler will function correctly as long as it receives a guaranteed amount of CPU time during each quantum. This can be accomplished by assigning to the PS scheduler a continuous CPU reservation with period equal to the quantum size, or scheduling it using a USP and treating the end of each period as a deadline. □

This result is of limited use in real systems because the subdivision of scheduling quanta will result in fine-grained time slicing among different reservations or USPs. The purpose of scheduling quanta is to limit the number of context switches in a system. Therefore, fine-grained time slicing defeats the purpose of scheduling quanta.

5.3.1.5 CPU Reservations

Rez, the CPU reservation scheduler that was implemented for HLS, must be run at the root of the scheduling hierarchy unless it receives a RESU guarantee. It is capable of making use of such a guarantee because it is deadline-based, and consequently can make deadline information available to a uniformly slower processor scheduler.

[1]The variable-length quantum is a holdover from SFQ's origins in network scheduling, where quantum sizes are the times to transmit variable-length packets.

5.3.1.6 Time Sharing

Since a time-sharing scheduler does not make any guarantee to entities that it schedules, it can make use of any guarantee. More precisely, it requires a NULL guarantee, to which any other guarantee may be trivially converted.

5.3.1.7 Schedulers From the Literature

This section explains and justifies the guarantee conversions for the schedulers listed in the bottom part of Table 5.1.

The BSS-I [51] and PShED [52] schedulers provide uniformly slower processors to other schedulers. Any scheduler that has a run-time representation of its deadlines may use a uniformly slower processor.

The borrowed virtual time (BVT) scheduler [20] has not been shown to be capable of providing any reservation-like guarantee. It gives each thread a *warp* parameter that allows it to borrow against its future CPU allocation; this has the useful property of decreasing average dispatch latency for threads with high warps. Warp, in combination with two additional per-thread scheduler parameters *warp time limit* and *unwarp time requirement*, defeats PS analysis techniques that could be used to prove that BVT provides bounded allocation over a specific time interval. However, despite the additional scheduling parameters, BVT (1) limits the CPU allocation of each thread to its share (over an unspecified time interval) and (2) supports hierarchical scheduling in the sense that it only increases a thread's virtual time when it is actually running. Therefore, BVT is capable of splitting a PS guarantee into multiple PS guarantees.

The constant bandwidth server (CBS) [2] provides the same scheduling behavior as a basic, hard reservation. Since it is deadline based, it can be scheduled by a uniformly slower processor.

The earliest eligible deadline first (EEVDF) algorithm [85] was shown to provide proportional share scheduling with bounded error. It has not been shown to work when given less than the full processor bandwidth.

The Linux scheduler is a typical time-sharing scheduler that provides no guarantees.

Lottery and Stride scheduling [90] provide proportional share resource allocation but do not bound allocation error. They have been shown to work correctly in a hierarchical environment.

The scheduler in the Portable Resource Kernel [68] was designed to be run as a root scheduler, and can provide both hard and soft CPU reservations. Although it is based on rate monotonic scheduling, it must have an internal representation of task deadlines in order to replenish application budgets. Therefore, it could be adapted to be scheduled using a uniformly slower processor.

Rialto [40] and Rialto/NT [37] are reservation schedulers. They could be adapted to be scheduled using a USP because they have an internal representation of their deadlines.

SFS [15] is a multiprocessor proportional share scheduler. It does not provide bounded allocation error to entities that it schedules.

The Spring operating system [82] uses a deadline-based real-time scheduler to provide hard real-time guarantees to tasks that it schedules. Since it is deadline-based, it can make use of s USP.

The total bandwidth server (TBS) [79] provides soft CPU reservations: it guarantees that a minimum fraction of the total processor bandwidth will be available to entities that it schedules, but it can also take advantage of slack time in the schedule to provide extra CPU time. It is deadline-based, and consequently can be scheduled using a USP.

The Windows 2000 time-sharing scheduler requires and provides no guarantees.

ALL	t	t	f	t	f	t	t	f	f	t	t	t
RESU	f	t	f	f	f	f	f	f	f	f	t	t
RESBH	f	f	t	t	f, 5.3	t, 5.2	t	f	f	t, 5.5	t, 5.6	t
RESBS	f	f	f	t	f, 5.3	t, 5.2	t	f	f	t, 5.5	t, 5.6	t
RESCH	f	f	t	t	t	t	t	f	f	t, 5.4	t, 5.6	t
RESCS	f	f	f	t	f	t	t	f	f	t, 5.4	t, 5.6	t
RESPS	f	f	f	t	f, 5.3	t, 5.2	t	f	f	t, 5.5	t, 5.6	t
RESNH	f	f	t	t	t	t	t	t	f	t, 5.4	t, 5.6	t
RESSH	f	f	t	t	t	t	t	t	t	t, 5.4	t, 5.6	t
PSBE	f	f	f	t, 5.7	f	t, 5.7	t	f	f	t	t	t
PS	f	f	f	f	f	f	f	f	f	f	t	t
NULL	f	f	f	f	f	f	f	f	f	f	f	t
$\mapsto$	ALL	RESU	RESBH	RESBS	RESCH	RESCS	RESPS	RESNH	RESSH	PSBE	PS	NULL

Table 5.2: Guarantee conversion matrix

5.3.2 Converting Guarantees Using Rewrite Rules

Rewrite rules exploit the underlying similarities between different kinds of soft real-time scheduling. For example, it is valid to convert any CPU reservation with amount x and period y to the guarantee PS x/y. This conversion means that any pattern of processor allocation that meets the requirements for being a CPU reservation also meets the requirements for being a PS guarantee. Clearly, the reverse conversion cannot be performed: a fixed fraction of the CPU over an unspecified time interval is not, in general, equivalent to any particular CPU reservation.

Table 5.2 shows which guarantees can be converted into which others using rewrite rules. Characters in the matrix indicate whether the guarantees listed on the left can be converted to the guarantees listed on the bottom. Feasible conversions are indicated by "t," while impossible conversions are indicated by "f." When a conversion or lack of conversion is non-trivial, the accompanying number refers to a theorem from this section.

The following lemma will be used to establish an equivalence between basic and continuous CPU reservations.

Lemma 5.1. *Given a thread with guarantee* RESBH x y *or* RESBS x y*, every time interval* $(2y-x)$ *units long contains at least x units of CPU time. Furthermore, any time interval smaller than this is not guaranteed to contain at least x units of CPU time.*

Proof. According to the definition on Section 5.2.2.2, a basic CPU reservation contains at least x units of CPU time in each time interval $[t+iy, t+(i+1)y]$ where t is a time chosen by the reservation scheduler and i is an integer. We refer to each of these time intervals as a period of the reservation. To show that an arbitrary time interval of length $2y-x$ is sufficiently long to always contain x units of CPU time, we examine two cases, depicted in Figure 5.2. In the first case, the $2y-x$ interval overlaps three periods of the reservation, and therefore contains a complete period. Then, by the definition of basic CPU reservation, there must be at least x units of CPU time within this interval. In the second case, the interval $2y-x$ overlaps two periods of the reservation, and does not contain a complete period. Without loss of generality, assume that it is contained by time interval $[t, t+2y]$. By the definition of basic CPU reservation, this interval must contain at least $2x$ units of CPU time. Observe that there is not enough room inside of this interval $2y$ units long to contain both $2x$ units of CPU time and the arbitrary interval $2y-x$ units long unless at least x of the units of CPU time are contained in the arbitrary interval.

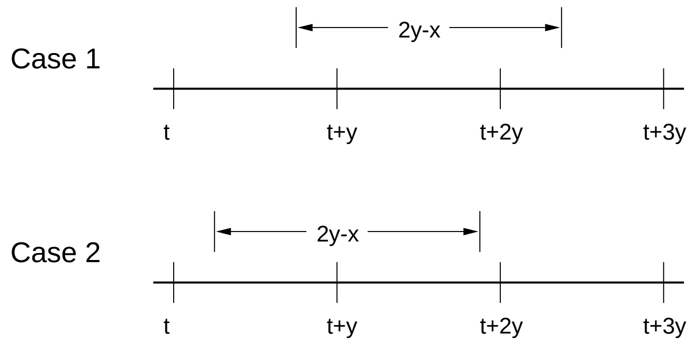

Figure 5.2: Time-lines for case analysis in the sufficient condition of the proof of Lemma 5.1

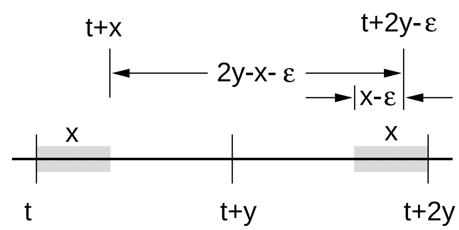

Figure 5.3: Time-line for the necessary condition of the proof of Lemma 5.1

We show that an interval of length $2y-x$ is necessary by contradiction. Assume that there exists an $0 < \varepsilon < x$ such that every time interval of length $2y-x-\varepsilon$ contains at least x units of CPU time. Without loss of generality we examine the time interval $[t, t+2y]$ where t is the time chosen by the scheduler in the definition of basic CPU reservation in Section 5.2.2.2. Furthermore, assume that the basic reservation scheduler allocates CPU time to the reservation during time intervals $[t, t+x]$ and $[t+2y-x, t+2y]$. In other words, it allocates CPU time at the beginning of the first period and the end of the second, as Figure 5.3 illustrates. Then during the time interval $[t+x, t+2y-\varepsilon]$ the reservation will have been scheduled for $x-\varepsilon$ time units, contradicting our assumption that every time interval $2y-x-\varepsilon$ units long contains at least x units of CPU time. □

The following theorem shows that basic CPU reservations can be converted into continuous CPU reservations.

Theorem 5.2. *The guarantees* RESBS x y *and* RESBH x y *can each be converted into the guarantee* RESCS x $(2y-x+c)$ *for any* $c \geq 0$.

Proof. Lemma 5.1 proved that an arbitrarily placed time interval $2y-x$ or more time units long will contain at least x units of CPU time. This meets the definition of a continuous CPU reservation. □

The following theorem proves that it is not possible to convert an arbitrary basic CPU reservation into a continuous, hard CPU reservation. The only basic CPU reservation that is also a hard, continuous CPU reservation is the trivial basic reservation that is equivalent to ALL.

Theorem 5.3. *Neither* RESBS x y *nor* RESBH x y *can be converted into a continuous, hard CPU reservation unless* $x = y$.

Proof. By Theorem 5.2 the guarantees RESBS x y and RESBH x y can each be converted into a continuous, soft CPU reservation with period x and amount $2y - x + c$ where $c \geq 0$. The soft CPU reservation cannot also be a hard reservation unless its utilization is bounded above as well as below. The requires that the utilization of the original basic reservation, which is x/y, be equal to the utilization of the new continuous reservation, which is $x/(2y - x + c)$. This condition is satisfied only when $x = y$ and $c = 0$. □

The next lemma will be used in the proof that continuous CPU reservations can be converted into proportional share guarantees with bounded error.

Lemma 5.2. *The longest time interval during which a thread with guarantee* RESCS x y *or* RESCH x y *can be starved is* $y - x$ *units long.*

Proof. A gap is a time interval during which a particular task is not scheduled. Assume that a task experiences a gap of length g, where $g > y - x$, starting at time t. Then, during the time interval $[t, t + y]$ the task will not be scheduled until time $t + g$, making it impossible for the task to receive x units of CPU time before time $t + y$. This is a contradiction according to the definition of continuous CPU reservation given in Section 5.2.2.2. □

The following theorem establishes a correspondence between continuous CPU reservations and proportional share guarantees with bounded error.

Theorem 5.4. *The guarantees* RESCH x y *or* RESCS x y *may be converted to the guarantee* PSBE $\frac{x}{y}$ $\frac{x}{y}(y - x)$.

Proof. The long-term average service rate of the CPU reservation is $\frac{x}{y}$. This is the correct value for the share part of the PS guarantee because share and long-term average service rate mean the same thing. δ is, intuitively, the farthest a thread receiving PS scheduling is allowed to fall behind its average service rate. By Lemma 5.2, the longest gap in service that a task with guarantee RESCS x y or RESCH x y can experience is $y - x$ units long. During a gap of length $y - x$ a reservation with service rate $\frac{x}{y}$ falls $\frac{x}{y}(y - x)$ units behind its average service rate. Therefore, this is the correct value for δ. □

The next lemma will be used to prove that basic CPU reservations can be converted into proportional share guarantees with bounded error.

Lemma 5.3. *The longest time interval during which a thread with guarantee* RESBS x y *or* RESBH x y *can be starved is* $2(y - x)$ *units long.*

Proof. A thread with a basic CPU reservation experiences the longest possible gap when the reservation scheduler schedules it at the beginning of one period and at the end of the next period. In other words, for a time t chosen by the scheduler, the application is scheduled during intervals $[t, t + x]$ and $[t + 2y - x, t + 2y]$. The gap in scheduling between these intervals is $2(y - x)$ time units long. □

The following theorem establishes a correspondence between basic CPU reservations and proportional share guarantees with bounded error.

Theorem 5.5. *The guarantees* RESBH x y *or* RESBS x y *may be converted to the guarantee* PSBE $\frac{x}{y}$ $2\frac{x}{y}(y - x)$.

Proof. The average service rate of a basic CPU reservation—like the average service rate of a continuous CPU reservation—is the same thing as the share part of a PS guarantee. By Lemma 5.3 the longest gap in service for an entity with a basic reservation is $2(y-x)$ units long. During the time interval $2(y-x)$, the service provided by the reservation will fall behind by $2\frac{x}{y}(y-x)$. Therefore, this is the correct value for δ. $\square$

The next theorem proves that any CPU reservation may be converted into a proportional share guarantee.

Theorem 5.6. *Any CPU reservation, whether hard or soft, basic or continuous, with amount x and period y may be converted into the guarantee* PS x/y.

Proof. The PS guarantee requires a specified long-term average fraction of the processor bandwidth. The long-term average fraction of the processor guaranteed by a CPU reservation is x/y. $\square$

The following theorem was motivated by an observation by Stoica et al. [84], who said it is possible to provide reservation semantics using a proportional share scheduler.

Theorem 5.7. *The guarantee* PSBE $s\ \delta$ *can, for any* $y \geq \frac{\delta}{s}$, *be converted into the guarantee* RESCS $(ys-\delta)\ y$ *or* RESBS $(ys-\delta)\ y$.

Proof. The proportional share guarantee states that during any time interval n units long, a thread with a share s of the CPU will receive at least $ns-\delta$ units of CPU time where δ is the allocation error bound. Let y be the period of the desired CPU reservation. A thread is then guaranteed to receive $ys-\delta$ units of CPU time during every time interval y units long. This is precisely the definition of a soft, continuous CPU reservation. Trivially, it is also a soft, basic reservation. y cannot be less than $\frac{\delta}{s}$ because that would make the quantity $(ys-\delta)$ negative. $\square$

5.3.3 Implications of the Equivalence of Reservations and Proportional Share with Bounded Error

Theorems 5.4, 5.5, and 5.7 show that CPU reservations can be converted into proportional share guarantees with bounded error and vice versa. The abstraction underlying all of these guarantees—a minimum allocation of CPU time over a specified time interval—is necessary for real-time scheduling to occur.

The conversions are useful when combined with the results that were proved about hierarchical SFQ schedulers. Recall that if a SFQ scheduler receives a PSBE guarantee it also provides this type of guarantee to its children. Therefore, a reservation scheduler can be used at the root of the scheduling hierarchy to schedule applications whose threads require precise or fine-grained scheduling, as well as giving a reservation to an instance of the SFQ scheduler. The reservation can be converted into a PSBE guarantee that is useful to the SFQ scheduler, which can then give performance guarantees to entities that it schedules.

There is a good reason not to use a proportional share scheduler as the root scheduler in a system, even though PS schedulers that have bounded allocation error are perfectly capable of providing CPU reservations. The reason is that while these schedulers bound allocation error, *they do not bound allocation error within a period of a thread's choosing*. To see this, let us first look at SFQ.

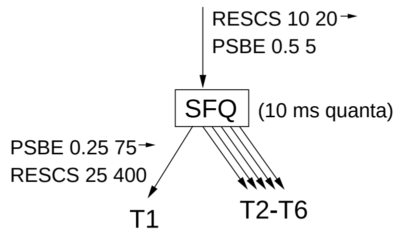

Figure 5.4: Using an SFQ scheduler to provide a CPU reservation

5.3.3.1 Providing a Reservation Using Start-Time Fair Queuing

Consider an SFQ scheduler that uses a quantum size of 10 ms (a common quantum size in a general-purpose operating system) and receives a guarantee of RESCS 10 20. It schedules 6 threads, T1-T6. T1 has weight 0.5 and T2-T6 each have weight 0.1. This scheduler is depicted in Figure 5.4. Applying Theorem 5.4, the SFQ scheduler's guarantee can be converted to the guarantee PSBE $\left(\frac{10}{20}\right)\left(\frac{10}{20}(20-10)\right)$, or PSBE 0.5 5. Using Equation 5.2, the thread with weight 0.5 can be shown to have a guarantee of PSBE $(0.5 \cdot 0.5)\left(0.5\frac{6 \cdot 10}{0.5} + 0.5\frac{5}{0.5} + 10\right)$, or PSBE 0.25 75. By Theorem 5.7, this guarantee can be converted to RESCS $(400 \cdot 0.25 - 75)$ 400, or RESCS 25 400.

This reservation is highly pessimistic: instead of being guaranteed to receive 25% of the CPU (its long-term average share), T1 is only guaranteed to receive 6.25% of the CPU during any given 400 ms period. In order to bring the level of pessimism down to 10% the reservation period y would have to be 3000 ms, or 3 s—far too long a deadline for most multimedia applications. This example illustrates a serious shortcoming of SFQ: it has delay bounds that are proportional to the number of threads that it is scheduling. In a typical time-sharing system with dozens of threads the kind of performance guarantee given to threads by an SFQ scheduler would not be useful for scheduling multimedia applications.

5.3.3.2 Providing a Reservation Using EEVDF

EEVDF [34] has a tighter error bound than SFQ: allocation error is bounded to the length of a single scheduling quantum. This bound is optimal in the sense that no quantum-based scheduler can have a lower error. To see the consequences of the EEVDF error bound, assume that a thread in a video application requires a guarantee of 5 ms / 33 ms in order to display 30 frames per second, and that we want to provide this reservation using an EEVDF scheduler. Assume that the EEVDF scheduler uses a 10 ms time quantum and is at the root of the scheduling hierarchy where it receives a guarantee of ALL.

The guarantee provided by the EEVDF scheduler will have the form PSBE s δ, where $\delta = 10$ ms. To construct a CPU reservation with period 33 ms from the PSBE guarantee we use Theorem 5.7, which states that the guarantee PSBE s δ can be converted into a soft, continuous CPU reservation with amount $(ys - \delta)$ and period y for any $y \geq \frac{\delta}{s}$. Combining this conversion with the PSBE guarantee, we can deduce that the reservation amount $(ys - \delta)$ will be $(33s - 10)$, which must equal 5 ms. This constrains the value of s to be $\frac{5+10}{33}$, or 0.45.

Therefore, providing a CPU reservation of 5 ms / 33 ms, or 15% of the CPU, using EEVDF requires allocating 45% of the processor bandwidth when the EEVDF scheduler uses a 10 ms quantum. Clearly, this degree of over-reservation is not acceptable in general. EEVDF's error bounds are optimal, so it is impossible for a different proportional share scheduler to do better than this. To reduce the level of pessimism, the only parameter available to adjust is the scheduler quantum size.

Assume that 10% is the largest degree of pessimism that is acceptable for the video player reservation. This means that s, the share of the CPU reserved for the thread, must be no more than $1.1(5/33)$, or 16.7% of the CPU. So, we assume that the EEVDF scheduler provides the guarantee PSBE 0.167 δ and we need to solve for a value of δ that allows the PSBE guarantee to be converted into the guarantee RESCS 5 33. To do this we apply Theorem 5.7 backwards: we know that x must be 5 ms and y must be 33 ms; this tells us that $5 = (33)(0.167) - \delta$. Solving for δ, we find that the required scheduling quantum is 0.5 ms.

Since EEVDF is an optimally fair PS scheduler, it will context switch between threads every 0.5 ms even when there are no real-time applications in the system. This will cause unacceptable inefficiency for threads that make use effective use of the cache. As we will show in Section 10.3.3, on a 500 MHz Pentium III a thread whose working set is 512 KB, the same size as the level-two cache, can take more than 2 ms to re-establish its working set in the cache after a context switch. Clearly, if context switches occur every 0.5 ms, applications with large working sets will not benefit from the cache.

These calculations indicate that proportional share schedulers are not suitable for providing CPU reservations to applications whose periods are of the same order of magnitude as the scheduler quantum size. We conclude that EDF- or static-priority-based reservation schedulers should be used to schedule applications with precise, fine-grained real-time requirements.

5.4 Conclusion

A hierarchy of schedulers composes correctly only if each scheduler in the hierarchy receives an acceptable guarantee, or a guarantee that can be converted into an acceptable guarantee. This chapter has (1) described the guarantee mechanism and how it can be used, (2) described the sets of guarantees that are acceptable and provided by a number of multimedia schedulers, and (3) shown which guarantees can be converted into which other guarantees, including proofs of the conversions when necessary.

Chapter 6

Issues and Examples for Scheduler Composition

The previous chapter described the *guarantee* mechanism for composing schedulers. This chapter presents several applications of scheduler composition. The first section describes how to compute the parameters for a CPU reservation that, when given to a rate monotonic scheduler, allows that scheduler to schedule all periodic threads in a multi-threaded real-time application. The second part of this chapter shows that complex idiomatic scheduling behavior can be composed using simple schedulers as components. The final section describes some additional guarantee issues, such as how multiprocessors are supported and limitations of the guarantee system.

6.1 Guarantees for Multithreaded Applications

Many multimedia applications are structured as groups of cooperating threads. For example, as Jones and Regehr [39] describe, when the Windows Media Player (the streaming audio and video application that ships with Windows systems) is used to play an audio file, it creates five threads with periods ranging from 45 ms to 1000 ms. One way to apply real-time scheduling techniques to such an application would be to assign a CPU reservation with appropriate period and amount to each thread in the application. Another way would be to assign a single CPU reservation to the entire application, and then use a rate monotonic scheduler to schedule the individual application threads. The resulting scheduling abstraction is similar to the multi-thread *activity* provided by Rialto [40], except that threads within a Rialto activity are scheduled earliest deadline first when they hold a time constraint, and round-robin otherwise. Also, no method was presented for calculating the amount of period of the CPU reservation that each Rialto activity should request. This section presents such a method.

Figure 6.1 shows a portion of a scheduling hierarchy that includes a per-application reservation. Threads T1, T2, and T3 all belong to the same application, and respectively require 1 ms of CPU time every 10 ms, 1 ms every 20 ms, and 5 ms every 100 ms. The fixed-priority scheduler FP schedules the threads rate monotonically by assigning priorities to the threads in order of increasing period: T1 gets the highest priority, T2 gets the middle priority, and T3 gets the lowest priority. Then, the rate monotonic scheduler is given a basic CPU reservation of 1.1 ms / 5 ms. The method that will be presented in Section 6.1.2 can be used to show that this reservation is sufficient to allow all three threads to meet their deadlines.

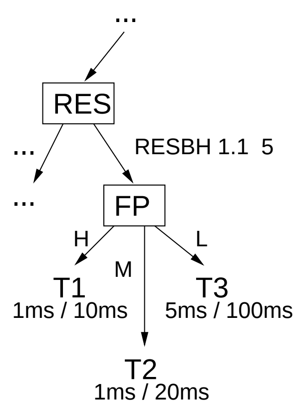

Figure 6.1: Example of a per-application guarantee. Threads T1, T2, and T3 all belong to the same application, and share a CPU reservation of 1.1 ms / 5 ms.

6.1.1 Pros and Cons

Reasons that a per-application reservation might be preferable to per-thread reservations include the following:

1. Since inter-thread isolation is no longer being enforced, applications have the freedom to internally reallocate CPU time if so desired.

2. Using the same reservation to schedule all application threads increases the likelihood that cooperating threads will be scheduled close to each other in time, increasing cache locality for data-intensive multimedia applications.

3. On a multiprocessor machine, a per-application reservation would cause all threads to run on the same processor (assuming that the reservation scheduler is pinned to a particular processor), further increasing cache locality and reducing expensive data migration between caches on different processors.

4. The application would become more resilient to transient increases in CPU requirements. These could be caused externally (for example, by a long-running interrupt handler stealing time from the application) or internally (by a slight change in application requirements). The added resilience to fluctuations in demand comes from the fact that since inter-thread isolation is no longer being enforced, slack time in the application's schedule is statistically multiplexed among all application threads.

5. If worst-case semaphore hold times can be calculated, more sophisticated variants of static-priority analysis can be used in order to guarantee schedulability for an application comprised of threads that synchronize with each other.

There are also several reasons why a per-application reservation might not be a good idea:

1. The rate monotonic analysis presented in this section is potentially more pessimistic than the EDF techniques that can be used to implement a reservation scheduler. In other words,

the per-application reservation may need as much as 31% (one minus the rate monotonic utilization bound of 69%) more CPU time allocated to it than the sum of the requirements of the individual threads.

2. Since all application threads run in the context of a single CPU reservation, applications will not be able to take advantage of thread-level parallelism in order to run faster on a multiprocessor.

3. The per-application reservation will tend to increase the number of context switches experienced by threads with relatively long periods. To see this, assume that an application is comprised of two threads: one that has a requirement of 1 ms / 10 ms and one that requires 100 ms / 1000 ms. If each were scheduled under its own reservation on an otherwise idle machine, the period 10 thread would never be preempted (or would be preempted seldomly), and the period 1000 thread would usually be preempted 10 times by the period 10 thread. If the two were scheduled using a per-application CPU reservation of 2 ms / 10 ms, then the short-period thread would run for 1 ms of each 2 ms time slice, and the long-period thread would run for the other 1 ms. Therefore, the number of preemptions experienced by the long-period thread would be increased by a factor of 10. This could impose a noticeable performance penalty if the long-period task used the cache effectively—increasing the number of preemptions means that it would potentially start out with a cold cache every millisecond.

A thorough investigation of the tradeoffs between per-thread and per-application reservations is beyond the scope of this work.

6.1.2 Theory

This section presents a method for calculating the parameters of a CPU reservation that, when provided to a rate monotonic scheduler, allows that scheduler to meet the CPU requirements of an application comprised of multiple periodic threads. The approach is to use static-priority analysis to test the schedulability of the group of threads comprising the application plus a *ghost thread* that represents the time not available to the application because the reservation scheduler is scheduling a different application. Since the ghost thread cannot be "preempted" by the application, it must have the highest priority in the analysis.

The following lemma establishes the equivalence between a continuous, non-preemptive CPU reservation and static priority scheduling including a ghost task. This equivalence is important because it will allow us to expand the domain of applicability of static priority analysis to hierarchical task sets.

Lemma 6.1. *Assume a thread T_1 is scheduled at low priority by a fixed priority scheduler that (1) is given the guarantee* ALL, *and (2) schedules at high priority a thread T_2 that has period y and amount $(y-x)$, and has no release jitter. Then, the scheduling behavior received by T_1 is the same as it would receive if it were given the guarantee* RESNH x y.

Proof. Let t_n denote the beginning of the nth period of T_2. Since T_2 has no release jitter, it always starts running at the beginning of its period and blocks at time $t_n + y - x$. This allows T_1 to run during the time interval $[t_n + y - x, t_n + y]$, giving x uninterrupted units of CPU time to T_1 at the same offset during each time interval y units long. This meets the definition of a non-preemptive hard CPU reservation with amount x and period y. □

Theorem 6.1. *If a static priority scheduler that is given the guarantee* ALL *can schedule the task set containing the threads of a multi-threaded application plus a "ghost" task with period y and amount $(y-x)$ that runs at the highest priority, then the application can also be scheduled using a rate monotonic scheduler that is given a continuous, non-preemptible CPU reservation with amount x and period y.*

Proof. Lemma 6.1 showed that a thread receives the same scheduling whether it is given the guarantee RESNH x y or scheduled by a fixed priority scheduler alongside a higher-priority "ghost task." Since the schedule produced in both situations is the same, it is the case that if all application threads meet their deadlines when scheduled by the fixed-priority scheduler, they will also meet their deadlines when scheduled by the CPU reservation. In other words, the choice of scheduling algorithm is irrelevant because we have shown that they produce equivalent schedules. □

The usefulness of Theorem 6.1 is limited by the fact that real schedulers cannot easily provide continuous, non-preemptible CPU reservations—the schedulability analysis for task sets containing non-preemptive tasks that require significant amounts of CPU time is very pessimistic. The following lemma will be used to support a theorem that relaxes the assumption of non-preemptive reservations.

Lemma 6.2. *Assume a thread T_1 is scheduled at low priority by a fixed priority scheduler that (1) is given the guarantee* ALL*, and (2) schedules at high priority a thread T_2 that has period y and amount $(y-x)$, and has release jitter of up to x time units. Then, the scheduling behavior received by T_1 is the same as it would receive if it were given the guarantee* RESBH *x y.*

Proof. Let t_n denote the beginning of the nth period of T_2. Since T_2 has x time units of release jitter, it may begin running at any time during the time interval $[t_n, t_n + x]$. However, regardless of when it starts, it will finish running $(y-x)$ time units later. Therefore, T_1 will always be able to run for x units of CPU time during each period y units long. This schedule meets the definition of a basic, hard CPU reservation with amount x and period y. □

Theorem 6.2. *If a static priority scheduler that is given the full CPU bandwidth can schedule the task set containing the threads of a multi-threaded application plus a "ghost" task with period y and amount $(y-x)$ that runs at the highest priority and that also has release jitter allowing it to run at any offset within its period, then the application threads can also be scheduled using a rate monotonic scheduler that is given a basic CPU reservation with amount x and period y time units.*

Proof. Lemma 6.2 showed that a thread receives the same scheduling whether it is given the guarantee RESBH x y or scheduled by a fixed priority scheduler alongside a higher-priority "ghost task" that has release jitter, allowing it to run at any offset during its period. Since the schedule produced in both situations is the same, it is the case that if all application threads meet their deadlines when scheduled by the fixed-priority scheduler, they will also meet their deadlines when scheduled by the CPU reservation. □

To apply Theorem 6.2 in practice, we use a version of static priority analysis that takes release jitter into account [87]. Release jitter gives a task the freedom to be released, or to become ready to run, at times other than the beginning of its period. For example, a task with release jitter could wait until the end of one period to run and then run at the beginning of the next period—the possibility of "back-to-back" task occurrences of course has a negative influence on the schedulability of any lower-priority tasks. The analogous situation for a thread scheduled by a basic reservation scheduler happens when it is scheduled at the beginning of one period and the end of the next.

Static priority analysis is a generalization of rate monotonic analysis that does not require thread periods and priorities to be coupled. The freedom that a basic reservation scheduler has to decide when, within a task's period, to allocate CPU time to a task is modeled in the fixed priority analysis by giving the ghost task the freedom to be released at any time during its period. To decide whether an application task set in combination with a ghost task is schedulable, the worst-case response time of each task must be calculated. Since we assume that task deadlines are equal to task periods, the task set as a whole is schedulable if and only if the worst-case response time of each task is less than or equal to its period.

The following formulas and notation are from Tindell et al. [87]. For task i, let r_i be the worst-case response time, C_i be the worst-case execution time, T_i be period, and J_i be the release jitter. Also, let $hp(i)$ be the set of tasks with higher priority than i. Tasks are numbered from $1..n$ in decreasing order of priority. Task 1 is the ghost task, and its release jitter is $T_j - C_j$, the worst possible jitter that will still allow the ghost task to meet its deadline. Application threads $1..n$ are mapped to static-priority tasks $2..n+1$, and all application threads have zero release jitter. Then, the following formula

$$r_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + r_i}{T_j} \right\rceil C_j \tag{6.1}$$

can be used to find the worst-case response time of each task. Although r_i appears on both sides of the preceding formula, the recurrence relation

$$r_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + r_i^n}{T_j} \right\rceil C_j \tag{6.2}$$

can be used to iteratively calculate response times where r_i^n is the nth approximation of r_i. The recurrence provably converges when r_i^0, the initial estimate of response time for task i, is zero and the utilization of the entire task set is less than 100%.

All parameters except for C_1 and T_1, the amount and period of the ghost task, are known. The problem is to find the largest possible amount and period for the ghost task that allows the combined task set to be schedulable. The period of the per-application reservation will then be T_1 and the amount will be the time not used by the ghost reservation: $T_1 - C_1$.

Lacking a closed-form solution to this problem, we solve it iteratively. Unfortunately, the solution is not well defined since there are two unknowns. One way to constrain the solution space would be to find the ghost reservation with the largest utilization (i.e., that maximizes $\frac{C_1}{T_1}$) that still allows the resulting task set to be schedulable, and then to search for the reservation having that utilization that has the largest period (larger periods are desirable since they reduce the expected number of unnecessary context switches). However, better heuristics may be possible: to reduce unnecessary context switches it may be acceptable to choose a ghost reservation with a slightly smaller utilization than the largest possible utilization if the period of the ghost reservation is significantly longer.

Thread	Period	Amount
1	10	1
2	45	2
3	100	4
4	100	3
5	500	7
6	2000	25

Table 6.1: Requirements of threads in the Windows Media Player. Total utilization is 24.1%.

6.1.3 Example

This section presents a concrete example of the calculation described in the previous section. The task set for the example was generated using the periods of the tasks in Windows Media Player [39]; the amount of CPU time required by each thread was fabricated. A task representing the Windows kernel mixer, a middleware thread that must also receive real-time scheduling for the Media Player to operate correctly, was added to the task set. All periods and amounts are listed in Table 6.1; their total utilization is 24.1% of the CPU.

Figure 6.2 shows the parameter space of CPU reservations for the Media Player task set. The figure was generated by testing the schedulability of the task set for each combination of reservation parameters. The shaded region indicates amount / period combinations that do not allow a rate monotonic scheduler to schedule the Media Player task set without missing any deadlines. A reservation of around 1.5 ms / 6 ms, chosen from near the inflection point on this graph, would be ideal to assign to the Media Player task set—making its period longer requires increased utilization and making its period shorter will incur extra context switch overhead without gaining any utilization.

The borders between the schedulable and unschedulable regions are roughly defined by the line between reservations with utilizations greater than and less than 24.1% (the line with a shallow slope), and the line between reservations with gaps longer and shorter than 9 ms (the line with steeper slope). Clearly, no reservation with utilization less than the utilization of the task set can guarantee its schedulability. The cause of the second line is less obvious—the "gap" in a reservation is the longest time interval that may not give any CPU time to the application being scheduled. To see why a basic CPU reservation of 1.5 ms / 6 ms has a maximum gap of 9 ms, observe that the reservation scheduler could schedule the application at the beginning of one period (leaving a 4.5 ms gap until the end of the period) and the end of the next period (leaving a second 4.5 ms gap). The total gap is then 9 ms long, which barely allows the Media Player thread with amount 1 and period 10 to meet its deadline (by being scheduled on either side of the gap). A heuristic search through the two-variable parameter space can be used to find a reservation assignment that is optimal in the sense that reserving either more time or at a shorter period would not allow that application to meet all deadlines.

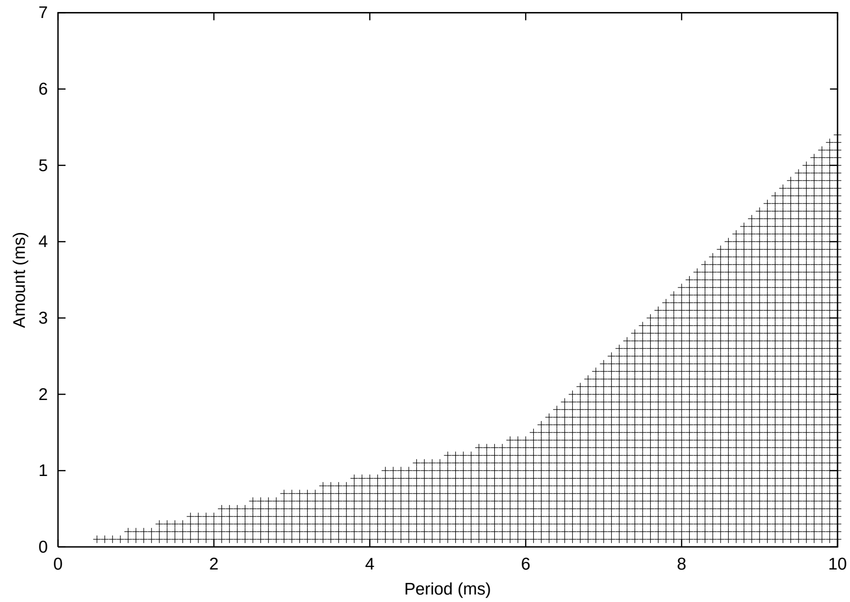

Figure 6.2: Parameter space for a per-application CPU reservation. The shaded region indicates period / amount combinations that are not sufficient to schedule the Media Player task set. All CPU reservations in the unshaded region can successfully schedule the task set.

6.2 Synthesizing Complex Behaviors from Simple Schedulers

Schedulers presented in the literature often provide scheduling behavior that is more complex than simple CPU reservations or proportional share scheduling, based on the hypothesis that certain complex scheduling behaviors are a good match for the kinds of overall scheduling behavior that real-world users and applications require.

This section demonstrates that a variety of complex schedulers can be synthesized from simple components, with the guarantee system helping to ensure correct composition. The fundamental insight is that scheduling policies can be implemented as much by the shape of the scheduling hierarchy as they are by schedulers themselves.

There are several reasons to build a complex scheduler on the fly from hierarchical components rather than implementing it as a fixed part of an operating system. First, modular schedulers can be more easily extended, restricted, and modified than monolithic schedulers. Second, the overhead associated with complex scheduling behaviors need only be incurred when complex behavior is required—complex arrangements of schedulers can be dismantled as soon as they are not needed. Finally, different complex scheduling behaviors can be combined in the same system. For example, the *open system architecture* defined by Deng et al. [18] provides support for multi-threaded real-time applications. However, the open system performs schedulability analysis based on worst-case execution times, making it difficult to schedule applications such as MPEG decoders whose worst-case CPU requirements are much larger than their average-case requirements. *Probabilistic CPU reservations* [16] were designed to solve exactly this problem. However, implementing them in the open system architecture (or vice versa) would be a difficult, time-consuming task. Using HLS, the

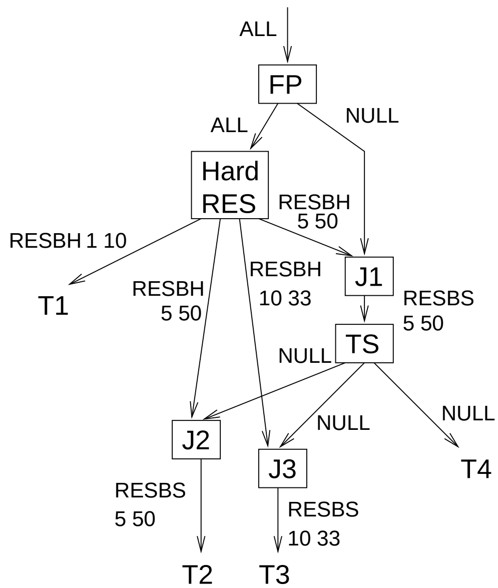

Figure 6.3: Example of a scheduling hierarchy that provides a hard reservation (T1), soft reservations (T2 and T3), and time-sharing behavior (T4). Arcs are labeled with guarantees.

two behaviors could be easily combined.

6.2.1 Hard, Firm, and Soft CPU Reservations

The implementors of schedulers providing CPU reservations have found it useful to characterize reservations as soft, guaranteeing that an application will receive a minimum amount of CPU time, or hard, guaranteeing a maximum as well as a minimum. Some multimedia applications limit their own CPU usage—at the beginning of each period they perform work and then block until the start of the next period. For these applications, it is irrelevant whether the guarantee they receive is hard or soft. For other applications, a soft CPU reservation may be useful if they can provide added value given extra CPU time. Hard reservations can be used to limit the processor usage of threads that may overrun, or that are CPU-bound.

Rialto [40], Rialto/NT [37], and the total bandwidth server [79] provide soft CPU reservations. The Rez scheduler presented in Section 9.3.2 provides hard CPU reservations, as does the constant bandwidth server [2]. The portable resource kernel for Linux [68] provides hard and soft CPU reservations, in addition to “firm” reservations, a special case of soft reservations that only receive extra CPU time when no other entity in the system requests it.

Figure 6.3 shows a scheduling hierarchy that can provide hard, firm, and soft CPU reservations. Each arc on the graph is labeled with the guarantee that is being provided. A fixed-priority (FP) scheduler at the root of the hierarchy allows a scheduler that provides hard CPU reservations to run whenever it has something to schedule. The join scheduler J1 combines a hard CPU reservation for the time-sharing scheduler (to ensure fast response time for interactive tasks) with any CPU time not used by the reservation scheduler. Threads 2 and 3 have soft CPU reservations provided by adding time-sharing scheduling behavior to their respective join schedulers. If J2 were scheduled by the time-sharing scheduler at a very low priority, than it could be said to have a firm CPU reservation—it would not get extra CPU time unless no other time sharing thread were runnable. Finally, Thread 4 is a normal time-sharing thread.

In a system containing a resource manager, probabilistic CPU reservations would be granted using a custom rule that performs the following actions: first, it creates a join scheduler and arranges for it to receive time-sharing scheduling; second, it moves the requesting thread to the join scheduler and finally, it requests a CPU reservation for the join scheduler. If all of these steps are successful, a soft CPU reservation has been successfully granted. In a system lacking a resource manager, a library routine or script would be used to provide a probabilistic CPU reservations by performing the same set of actions using the `HLSCtl` interface.

6.2.2 Probabilistic CPU Reservations

Chu and Nahrstedt [16] developed a specialized variant of the CPU reservation abstraction that they called *CPU service classes*. We have been calling this scheduling abstraction *probabilistic CPU reservation* in order to make them correspond with the names of other kinds of CPU reservations. Probabilistic CPU reservations are intended to be used to schedule applications whose worst-case CPU utilizations are much larger than their average-case utilizations (for example, MPEG decoders). Each such application gets a CPU reservation that meets its average-case CPU requirement. Furthermore, all applications with probabilistic reservations share an *overrun partition*—an extra CPU reservation that acts as a server for threads whose demands exceed their amount of reserved CPU time. Since each application is assumed to overrun only a small percentage of the time, the overrun partition is statistically multiplexed among all probabilistic CPU reservations. When demand for the overrun partition collides, the requirements of some applications will not be met.

Figure 6.4 shows a scheduling hierarchy that provides probabilistic CPU reservations. Thread 1 has a hard CPU reservation. Threads 2 and 3 are video decoders whose average-case CPU requirements are is 5 ms / 33 ms, and whose maximum CPU requirements are 15 ms / 33 ms. To implement probabilistic reservations, the threads share an overrun partition OVR that has reserved 10 ms / 33 ms. The desired behavior is for each of Threads 2 and 3 to be scheduled from the overrun partition only when they have exhausted the budgets provided to them by the reservation scheduler (that is, when a thread has already run for 5 ms during a 33 ms period, causing the reservation scheduler to refuse to schedule it until the next period begins). To accomplish this, extra information must be passed between the join schedulers and the reservation scheduler. When the reservation scheduler revokes the CPU from a join scheduler (join schedulers were described in Section 5.3.1.2), it also includes extra information notifying the join scheduler about why the processor being revoked: possible reasons are (1) the reservation budget has been exhausted and (2) the reservation scheduler has simply decided that another thread is more urgent. Only in the first case should the join scheduler then request scheduling service from the overrun scheduler. Since the overrun scheduler makes no additional guarantee to applications, it can use any scheduling algorithm. EDF would be the best choice in the sense that it would maximize the chances that overrunning applications still meet their deadlines, but a round-robin scheduler could also be used.

Like requests for firm and soft CPU reservations, requests for probabilistic CPU reservations could be granted either by loading appropriate rules into the resource manager or by directly manipulating the scheduling hierarchy with the `HLSCtl` interface.

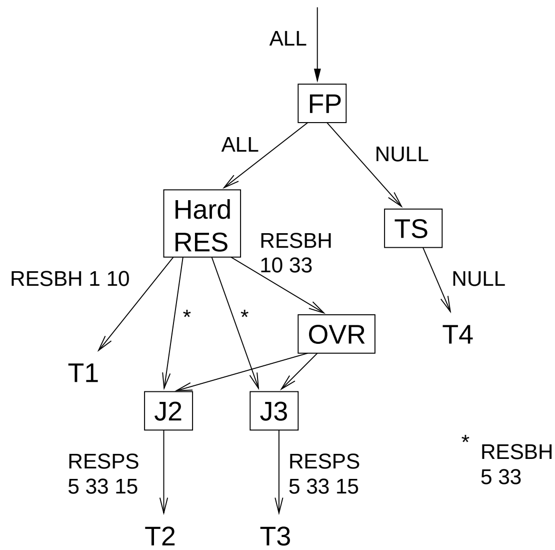

Figure 6.4: A scheduling hierarchy that provides probabilistic CPU reservations. Thread 1 has a hard CPU reservation, while threads 2 and 3 have probabilistic CPU reservations guaranteeing a minimum of 5 ms / 33 ms with a probabilistic maximum of 15 ms / 33 ms. Thread 4 receives default time-sharing scheduling behavior.

6.2.3 Integrated Hard and Soft Real-Time in Spring

Kaneko et al. [42] describe a method for integrated scheduling of hard and soft real-time tasks using a single hard real-time task as a server for scheduling soft real-time multimedia applications, amortizing the overhead of Spring's heavyweight planning scheduler. To implement this using hierarchical scheduling we would put the hard real-time scheduler at the root of the scheduling hierarchy, with the multimedia scheduler at the second level.

6.2.4 Rialto

Jones et al. [40] developed a scheduler for the Rialto operating system that is designed to support multi-threaded real-time applications. It provides (1) continuous CPU reservations to collections of threads called activities, and (2) time constraints to individual threads, giving them guaranteed deadline-based scheduling. Time granted to an activity by a CPU reservation is divided among the activity's threads by a round-robin scheduler unless there are active time constraints, in which case threads with active constraints are scheduled earliest-deadline first. CPU time requested to fulfill a new time constraint is first taken from an activity's reserved time and then (on a best effort basis) from idle time in the schedule. Threads that block during reserved time are allowed to build up a certain amount of credit—they are then given a second chance to meet their deadlines using slack time in the schedule on a best-effort basis. Finally, any remaining free time in the schedule is distributed among all threads on a round-robin basis.

The degree to which a decomposed, hierarchical version of Rialto can be constructed is limited

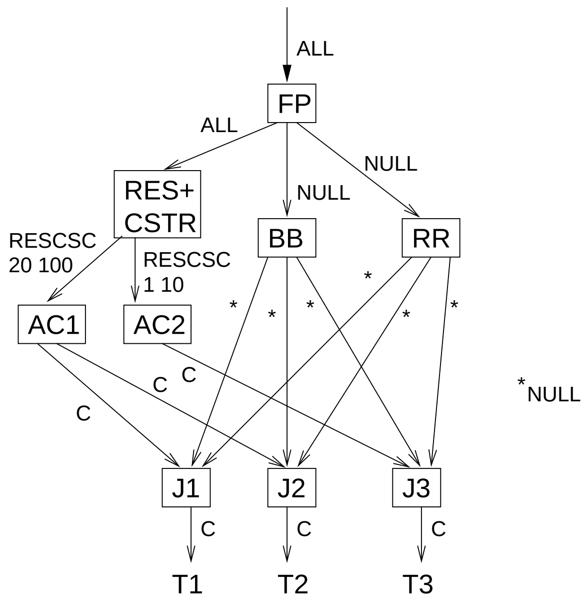

Figure 6.5: A scheduling hierarchy implementing functionality equivalent to the Rialto scheduler developed by Jones et al. [40]. A composite scheduler, RES+CSTR, manages the time line for CPU reservations and time constraints. A briefly-blocked scheduler (BB) performs best effort scheduling of that have built up credit due to blocking. A round-robin scheduler allocates any remaining CPU time.

by its integrated approach to managing the time line: CPU reservations and time constraints must share the data structures that represent particular reserved or unreserved time intervals. Furthermore, since time constraints effectively share the pool of unused time in the schedule, it would be difficult to provide a separate constraint scheduler for each activity.

A hierarchical scheduler equivalent to Rialto could be implemented by putting a static-priority scheduler at the root of the scheduling hierarchy that schedules a hybrid reservation/constraint scheduler at the highest priority, a "briefly blocked" scheduler at the middle priority, and finally a round-robin scheduler at the lowest priority. Figure 6.5 illustrates this arrangement. The composite RES+CSTR scheduler manages the time line for reservations and constraints; it schedules an activity scheduler (AC) at any time that it is scheduled either by a reservation or constraint. The RES+CSTR scheduler must also pass extra information to the activity scheduler informing it if a thread is currently being scheduled under a time constraint and, if so, which thread. An activity scheduler always schedules a thread that is running under a time constraint if there is one, and otherwise schedules threads in the activity in a round-robin manner.

The briefly blocked scheduler (BB) records the time at which threads block and unblock, in order to keep track of how much credit each thread has built up. When it gets to run, it picks a thread that has built up credit and schedules it. When there are no briefly blocked threads to run, the round-robin scheduler picks a thread to run and runs it. The 3-way join scheduler for each thread simply runs the thread whenever it is scheduled by any higher-level scheduler. Since Rialto was designed to schedule uniprocessors the case where a join scheduler is scheduled by two schedulers at once does not have to be addressed.

RESCSC and C represent hypothetical guarantees that could be integrated into HLS to support Rialto-style scheduling. RESCSC is a guarantee for a continuous, soft CPU reservation plus time constraints. Since activity schedulers schedule threads in a round robin manner, threads do not receive any guarantee unless they have an active time constraint, indicated by the C guarantee.

To see the benefits of the decomposed version of Rialto, notice that Rialto requires threads within an activity to use time constraints to meet their deadlines since the round-robin activity scheduler is not a real-time scheduler. Since using time constraints may impose a significant burden on the application developer, it may be preferable in some instances to instead schedule the threads in an activity using a rate monotonic scheduler according to the analysis presented in Section 6.1. To accomplish this, a particular activity would simply need to use a rate monotonic scheduler for its activity scheduler instead of the default Rialto activity scheduler. In fact, the activity schedulers can be replaced with arbitrary application-specific scheduling algorithms: the schedulability of other tasks cannot be adversely affected since the reservation scheduler isolates activities from each other. Also, to support applications that can opportunistically make use of extra CPU time to provide added value, the round-robin idle time scheduler could be replaced with a proportional share scheduler, allowing slack time to be preferentially allocated to applications that can best make use of it. These changes, which would likely require significant implementation effort to make to the original Rialto scheduler, could be made to a hierarchical version of Rialto in a straightforward manner.

6.3 Other Topics

6.3.1 Guarantees and Multiprocessors

Although the hierarchical scheduling architecture supports multiprocessors, the guarantee paradigm does not include explicit multiprocessor support because guarantees are made to schedulers through virtual processors, which by definition can make use of only one processor at a time. Multiprocessor guarantees can be made by providing more than one uniprocessor guarantee.

Since it provides a layer of indirection between applications and schedulers, the resource manager can be used to provide a unified front end to multiple uniprocessor schedulers, effectively merging them into a single multiprocessor scheduler. For example, a reservation scheduler can be instantiated once for each processor on a multiprocessor machine. Since each instance provides guarantees of the same kind, in absence of other constraints the resource manager will ask each scheduler in turn if it can grant a reservation, hiding the fact that there are multiple schedulers from real-time applications. Thus, in this case, the benefits of a multiprocessor reservation scheduler can be attained without actually having to write one. Furthermore, the resource manager can implement useful high-level policies such as putting reservations with similar periods on the same processor, reducing the number of unnecessary context switches. It does not make sense to provide this layer of indirection for a time-sharing scheduler, since time-sharing threads need to be able to move freely between processors in order to balance load.

To support both real-time applications and applications that require the services of a gang scheduler (to concurrently provide CPU time on multiple processors), a tradeoff must be made between the needs of the two different kinds of applications. In other words, a parallel application that cannot make rapid progress without the use of all four processors on a multiprocessor will conflict with a multimedia application that has a CPU reservation on one of the processors: during time reserved by the multimedia application, the parallel application is not able to make progress on any of the four processors. For any particular machine, this conflict is likely to be solved by assumption: either the machine will be part of a dedicated cluster, in which case real-time multimedia is unimportant, or it will be a desktop machine that must perform multimedia computations, in which case any background workload cannot assume reliable simultaneous access to multiple processors.

6.3.2 Integrating a New Scheduler into HLS

Integrating a new continuous media scheduler into the HLS framework is not expected to be difficult, as long as whatever guarantee the scheduler makes can be expressed in terms of an existing guarantee. This should be the common case since the existing guarantees cover a wide variety of multimedia schedulers that have appeared in the literature. If a new scheduler does not match any existing type but still fits into the guarantee paradigm (that is, it makes some sort of ongoing guarantee), then a new guarantee type needs to be constructed and rules for converting between its guarantees and those provided by existing schedulers must be written.

6.3.3 Limitations of the Guarantee Paradigm

Guarantees are static resource reservations that can be used to reason about the composition of schedulers as well as helping to match application requirements to the allocation patterns that schedulers provide. This section discusses some limitations of this paradigm.

6.3.3.1 Guarantees are Relative to CPU Speed

A precise guarantee such as a CPU reservation allows an application to be scheduled at a specified rate and granularity. Unfortunately, for a program running on a general-purpose operating system on a modern microprocessor, it can be difficult to map this rate and granularity into a metric that is useful for actual programs, such as a guarantee that a program will be able to get a specified amount of work done—for example, displaying one frame of video or decoding one buffer's worth of audio data before a deadline.

There are several reasons for this difficulty. First, there is substantial variation in the processors used in personal computers. There are several major CPU manufacturers for personal computers, and each of them has several models of CPUs, sometimes with different available features (floating point, MMX, etc.). Even when machines are identical at the instruction set level, there are different sizes and speeds of caches, numbers of TLB entries, lengths of pipelines, strategies for branch prediction, etc. Second, computer systems differ widely outside of the CPU: disks have different speeds, memory has different latency and bandwidth characteristics, and different I/O bus implementations have widely varying performance characteristics. Also, the overall performance of an application may be heavily influenced by the middleware and device drivers that the application is dynamically bound to. For example, the CPU usage of a game will be dramatically higher when it uses a graphics library that implements all 3D primitives in software than it will when bound to a library that makes use of a powerful hardware graphics accelerator. Furthermore, the performance of complex software artifacts such as graphics libraries can be difficult to characterize because certain common usage patterns are highly optimized compared to other (perfectly valid) usage patterns. Finally, complex applications are not amenable to worst-case execution time analysis. In fact, since they are implemented in Turing-complete languages, it is doubtful that many complex applications can be shown to have *any* bound on run time, much less a bound that is tight enough to be useful in practice. All of these factors, taken together, imply that it is very difficult to predict application performance in advance. Section 8.5.1 presents a possible method for determining application requirements in the context of a particular hardware and software environment.

6.3.3.2 Applicability of Guarantees

Because guarantees describe lower bounds on the amount of CPU time that threads will receive, their applicability for describing the scheduling behavior of complex, dynamic schedulers is limited. This is not a flaw in the guarantee model so much as a consequence of the fact that these schedulers make weak or nonexistent guarantees to entities that they schedule.

Time-sharing schedulers such as the default schedulers in Linux and Windows 2000 provide no real guarantees. And yet they are still useful—millions of people run multimedia applications on these operating systems using the techniques described in Section 3.4. Other classes of schedulers that provide no guarantees to applications include feedback- and progress-based schedulers, hybrid real-time schedulers such as SMART [67], and modified proportional share schedulers like BVT [20] and the modified Lottery scheduler developed by Petrou et al. [71]. The common theme across all of these schedulers is that they attempt to provide high value across the set of all running applications by dynamically allocating CPU time to where it is believed to be needed most. Because these schedulers retain the freedom to dynamically reallocate CPU time, they cannot make strong guarantees to applications: the two are mutually exclusive.

Schedulers that do not make guarantees to applications implicitly assume that applications' value degrades gracefully when their CPU requirements are not met. For example, they assume that if an application is given 70% of its CPU requirement, the application will provide approximately 70% of its full value. A program that performs real-time computer vision processing of a user's eye movements might degrade gracefully: when given less CPU than it requires, cursor movements will become jumpy, but they will still track eye movements. On the other hand, computer music synthesis and CD burning applications do not degrade gracefully. If they are given slightly less than their full requirements, they will sound bad and ruin discs, respectively, providing the user with considerably less than their full value.

Since both kinds of applications (gracefully and non-gracefully degrading) may be present in a single system, both should be supported. To accomplish this, a traditional EDF- or RM-based reservation scheduler could be used to provide precise guarantees to applications that do not degrade gracefully. To schedule the remaining applications, one of these two methods could be used:

1. Schedule time-sharing applications using a traditional time-sharing scheduler that has been assigned a CPU reservation with a small enough period that time-sharing applications are guaranteed to remain responsive to user input. If PS scheduling is also desired (for applications that require an ongoing share of the CPU but do not need hard bounds on latency), then a CPU reservation can be assigned to the PS scheduler as well.

2. Rather than running a traditional time-sharing scheduler, a proportional share scheduler with time-sharing extensions such as the one developed by Petrou et al. [71] could be used to schedule both time sharing and gracefully degrading multimedia applications.

6.3.3.3 Dealing with Dispatch Latency and Stolen Time

For threads to be able to receive the scheduling properties that schedulers guarantee, the operating system must not interfere with thread scheduling to the degree that guarantees are violated. Unfortunately, general-purpose operating systems can *steal* time from applications in order to perform low-level system activities. This time is referred to as stolen because thread schedulers in general-purpose operating systems are not aware of existence. In some cases, the effects of stolen time can be counteracted through over-reservation. For example, if an application requires 10% of the CPU during each 500 ms, than it will have a higher probability of making each deadline in the presence of stolen time if it is assigned a reservation of 15% per 500 ms than if it is assigned a reservation of 11% per 500 ms. However, as Jones and Regehr showed [38], the distribution of stolen time may not follow one of the usual statistical distributions. They described a system that experienced little stolen time until the network interface card was unplugged from the network, at which point a buggy network driver stole 19 ms every 10 seconds. In this situation it is impossible for an application thread with a period shorter than 19 ms to avoid missing a deadline, no matter how much it has over-reserved. Chapter 11 describes two schedulers that provide increased predictability in the presence stolen time.

6.4 Conclusion

This chapter has addressed practical issues in guarantee composition. First, we presented a method for scheduling a collection of cooperating threads using a rate monotonic scheduler that is given a CPU reservation. Second, we showed that several complex scheduling behaviors that have appeared in the literature can be composed using a number of small loadable schedulers as basic components. And finally, additional issues and limitations of the guarantee paradigm were discussed.

Chapter 7

Scheduling the Application Scenarios

This chapter solves the scheduling challenges from the application scenarios that were presented in Chapter 3. Each application scenario, on its own, does not constitute a strong motivation for the flexibility provided by HLS, since a monolithic scheduler could be constructed that has equivalent functionality and (most likely) is more efficient. The motivation for HLS comes from the lack of overlap among the scenarios—if the scheduling hierarchy for any of the scenarios was used to run any of the others, it would not be possible for applications to obtain the desired scheduling properties. In principle, a monolithic scheduler that provides the scheduling properties of all three hierarchies could be developed.[1] However, such a scheduler could easily turn into an engineering nightmare, and it would still be inflexible in face of the need for a new kind of scheduler, such as a gang scheduler.

7.1 A Machine Providing Virtual Web Servers

Since no real-time scheduling is required, this scenario can be scheduled by a homogeneous hierarchy like the one shown in Figure 7.1. Each scheduler in the hierarchy is a proportional share scheduler. At least PS1, PS5, and PS7 must be instances of a multiprocessor PS scheduler; for example, a multiprocessor version of BVT [20] or SFQ [27]. The scheduler should be augmented with a tunable processor affinity heuristic such as the one described by Chandra et al. [15]. The heuristic should be tuned for strong processor affinity to minimize the number of times threads migrate between processors, since throughput is a first-order concern in a web server. Also, a long scheduling quantum should be selected in order to allow most requests to complete within a single time quantum, minimizing the number of nonessential context switches.

In the hierarchy depicted in Figure 7.1, virtual server 1 (VS1), which is scheduled by PS2, receives a guarantee of PS 0.4, or 20% of the capacity of the two-processor machine. Since its share is less than 1, only one virtual processor is necessary to provide the CPU to PS2. However, since virtual server 2 (VS2) may be idle at times, it is advantageous to give VS1 two virtual processors in order to allow it to take advantage of both processors when VS2 is using less than its share. Rather than giving VS1 two guarantees of type PS 0.2, it is given a single guarantee of PS 0.4 to allow it to

[1] In fact, this could be “trivially” accomplished by adding code implementing all three monolithic schedulers into the kernel, creating a “super-monolithic” scheduler, and then providing a system call to switch between the behaviors.

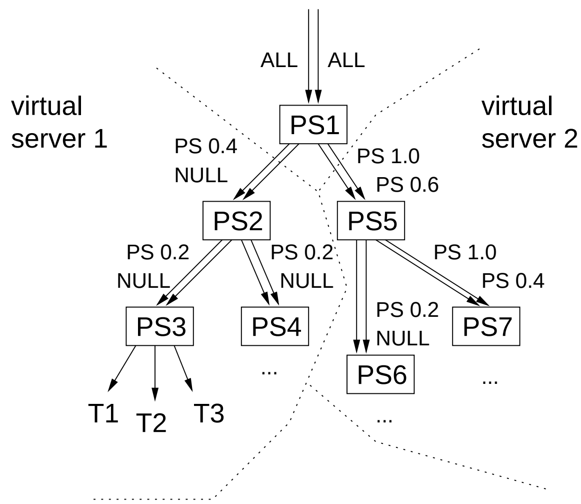

Figure 7.1: Example scheduling hierarchy for a two-processor machine providing two virtual servers that are divided by the dotted lines

run on the same processor as often as possible. Within VS1, schedulers PS3 and PS4 each receive an equal share of the processor time allocated to VS1. VS2 receives guarantees of PS 1.0 and PS 0.6, or 80% of the total capacity of the machine. It allocates guarantees of PS 1.0 and PS 0.4 to PS7, and PS 0.2 to PS6.

7.2 A Home Computer

A home computer must avoid starving time-sharing applications while supporting two classes of real-time applications. First, those that require an absolute share of the processor at a specific granularity since they provide little or no value when they miss deadlines. Second, those whose value degrades gracefully when they receive less CPU time than their full requirement.

A scheduling hierarchy that can meet the needs of these three classes of applications is shown in Figure 7.2. RES provides a basic, hard CPU reservation to real-time thread T1. Threads T2 and T3 are both part of the same application, and are being scheduled by a rate monotonic scheduler that has been given a CPU reservation (through join scheduler J2, which turns the hard, basic reservation into a soft, basic reservation) whose parameters have been calculated using the method from Section 6.1.

The reservation scheduler gives a reservation to the start-time fair queuing scheduler SFQ through join scheduler J1. J1 schedules SFQ during its CPU reservation and also during any time left idle by RES, resulting in a soft, basic CPU reservation of 10 ms / 20 ms. Theorem 5.5 shows how to convert a soft, basic CPU reservation into a proportional share guarantee with bounded error. This hierarchy makes use of this theorem to convert the soft CPU reservation into a guarantee that the SFQ scheduler can use. The SFQ scheduler, in turn, provides bounded-error guarantees to the time sharing scheduler and two gracefully degrading real-time threads T7 and T8, which would represent applications such as those that perform voice recognition or display stored video.

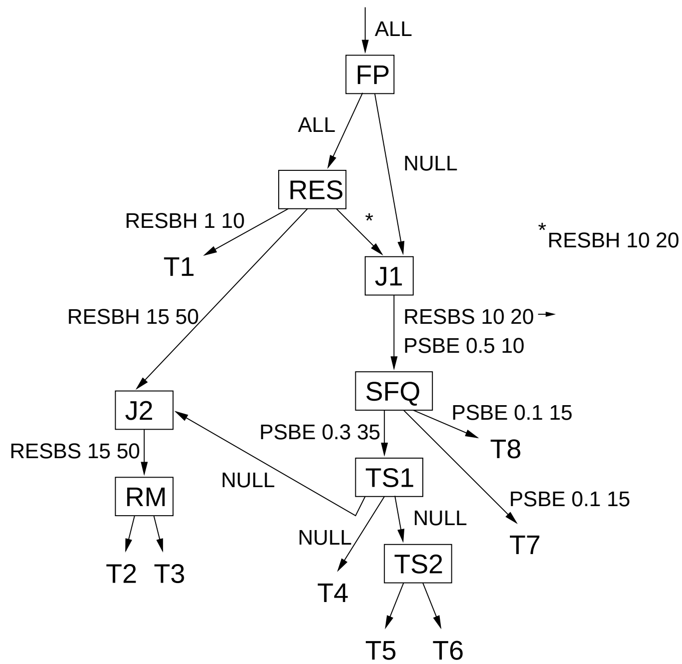

Figure 7.2: Example scheduling hierarchy for a home computer

The parameters of the proportional share guarantees provided by SFQ were calculated using Equation 5.2. The guarantee received by the time sharing scheduler is PSBE 0.3 35, which means that its minimum guaranteed share of the CPU over a long time period is 0.3, and that during no time interval will its allocation fall more than 35 ms behind what it "should" have received. This corresponds to maximum starvation period of roughly 100 ms for time-sharing applications, which is acceptable.

The differing admission policies provided by RES and SFQ can be seen by looking at what happens when extra threads request service. Since 90% of RES's overall capacity is reserved (10% for T1, 30% for J2, and 50% for J1), a new request for a CPU reservation with utilization more than 10% would have to be rejected. However, since SFQ is acting as a best-effort scheduler for applications whose performance gracefully degrades, it can grant any number of new requests. Of course, each new request that it grants will change the guarantees received by T7, T8, and TS1. The time-sharing scheduler TS1 can also admit any number of new threads, since it makes no guarantees. Time sharing scheduler TS2 represents a scheduler that is running an untrusted application that has created threads T5 and T6. The reason for adding an extra level to the scheduling hierarchy for an untrusted application is that no matter how many threads it creates, they will all be scheduled by TS2, which receives only a single thread's worth of CPU allocation from TS1.

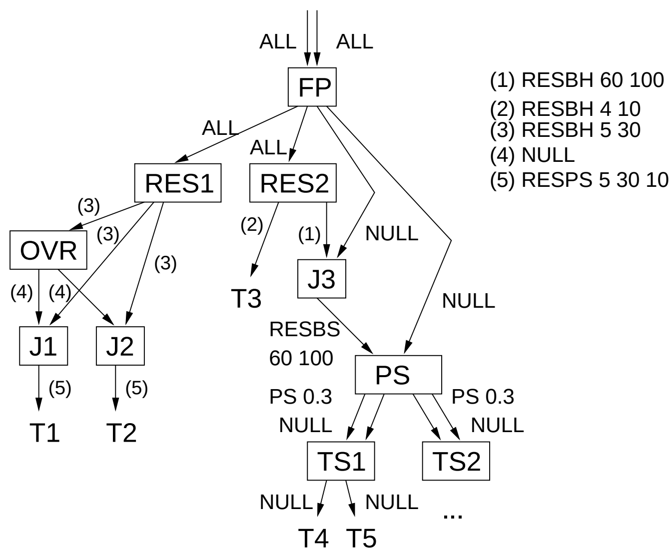

Figure 7.3: Example scheduling hierarchy for a terminal server

7.3 A Corporate or Departmental Terminal Server

A scheduling hierarchy that could be used to solve the scheduling challenges posed by this scenario on a two-processor machine is depicted in Figure 7.3. In this example only two users are logged in to the terminal server. The relevant features of this hierarchy are:

- A uniprocessor reservation scheduler is instantiated for each processor (RES1 and RES2).
- RES2 gives a guarantee of 60 ms / 100 ms to PS through J3. This guarantees that the time-sharing class of applications will always receive at least 60% of a processor.
- TS1 is the default scheduler for user 1, who is running two threads, T4 and T5. TS2 is the default scheduler for user 2. All time-sharing threads belonging to each user are scheduled by the corresponding time-sharing scheduler; this provides load isolation between users. Each user is guaranteed to receive 30% of a processor over a time interval selected by the PS scheduler.
- T3, a real-time thread belonging to user 1, has reserved 40% of a processor.
- Each of the two users is running a video player application; these are represented by threads T1 and T2. Because the worst-case CPU utilization of a video application can be worse than its average case usage, they are good candidates for probabilistic CPU reservations. Threads T1 and T2 share the use of an overrun reservation OVR, using the method that was described in Section 6.2.2.

A refinement of this scheduling hierarchy would be to dynamically change the weights assigned to TS1 and TS2 in order to grant each user a constant share over the entire scheduling hierarchy.

For example, at the same time that user 1 is granted a CPU reservation with utilization 10%, the amount of time guaranteed to her time-sharing scheduler would be decreased by 10%. This would eliminate a performance isolation loophole in the hierarchy in Figure 7.3 that allows users with real-time threads to receive more CPU time than users who only have time sharing threads.

7.4 Conclusions

Using three example scenarios, this chapter has shown that HLS can support combinations of applications with complex scheduling requirements, as well as enforcing isolation between resource principals at multiple levels.

Chapter 8

The Resource Manager

This chapter presents the design of the *resource manager*, a middleware application that adds value to the scheduling hierarchy by permitting guarantees to be reasoned about dynamically using the guarantee system that was described in Chapter 5, in addition to a rule-based system that can enforce high-level rules about the allocation of processor time.

There is a synergy between the scheduling hierarchy and the resource manager—each has capabilities that the other does not. For example, the resource manager, on its own, cannot provide any particular kind of scheduling behavior at all. The scheduling hierarchy, on its own, cannot limit the aggregate CPU usage of a user across all schedulers.

8.1 Introduction

The resource manager is an organized collection of hooks and reflective information that user-supplied code can use to make resource allocation decisions. The primary goal of the resource manager is: for a given set of requirements supplied by applications, and rules supplied by users and system administrators, to generate a mapping of threads to schedulers in that hierarchy that violates no rules and satisfies as many application requirements as possible. It will not always be possible to meet all application requirements because they may result in overload. In general, administrator requirements enforce fairness between users (or other resource principals) and user requirements tell the system which application requirements should be met when there is a conflict between them.

The following list of assumptions is designed to limit the scope of the resource manager to a tractable set of features.

- *The resource manager is not a planner.* The resource manager has no internal representation of time and never attempts to anticipate future application requirements. Rather, it is purely reactive: each decision is made using currently available information.
- *Rules are trusted.* Resource principals such as users, applications, and administrative domains may cause code to be loaded into the resource manager that makes decisions on their behalf. This code is assumed to have been written by a trusted authority.
- *There are no built-in policies.* In its default mode, the resource manager simply passes application requests through to the scheduling hierarchy, where schedulers provide their native

resource allocation policies (best effort for time-sharing and proportional share schedulers and admission control for reservation-based schedulers). More complex policies (such as attempting to maximize the utility of the currently running set of applications) may be implemented by code loaded into the resource manager.

- *It is not acceptable to require applications to be modified in order to use the resource manager.* Few existing production multimedia applications are resource self-aware. Rather, they simply assume that their scheduling requirements will be met, and possibly shed load in order to catch up if they fall behind. To be useful in the short-term, a multimedia operating system must be able to perform real-time scheduling without requiring applications to be modified.

8.2 Entities in the Resource Manager

The following entities exist within the resource manager:

- **Resource principals** represent entities whose aggregate resource usage is to be controlled, such as users, administrative domains, or accounting domains. Resource principals are not hierarchical.

- **Guarantees** are held by each application thread and scheduler in the system. Multiprocessor schedulers can have multiple guarantees. Each guarantee is owned by a single resource principal. Application guarantees are always owned by the resource principal who ran the application. A *system* resource principal owns the ALL guarantee given to the root scheduler. The system also owns each guarantee given by a scheduler whose guarantee is owned by the system, unless a guarantee is explicitly *delegated* to a different resource principal. Guarantees belonging to resource principals other than the system can be freely sub-divided without consulting any other resource principal. No principal other than the system may *revoke* a guarantee belonging to any other principal; the system can revoke any guarantee.

- **Requests** for guarantees are made by applications and schedulers. Requests consist of a requested guarantee in addition to optional user-specified information that rules can use to interpret which requests should be granted in the case that not all of them can. Requests that are malformed syntactically or semantically (for example, by requesting 110% of the CPU) return an error code to the requesting program. Well-formed requests that cannot be granted may remain *pending*, to possibly be granted later. Representations of both granted and pending requests remain in the resource manager until the requesting application either *cancels* the request or exits.

- **Guarantee rewrite rules** are loaded into the resource manager along with loadable schedulers.

- **Loadable schedulers** are binary objects that are demand-loaded into the kernel address space. They include a representation of the guarantees that they require and that they can provide. Schedulers are not owned.

- **Scheduler instances** are arranged in a hierarchy (or more precisely, a directed graph). Scheduler instances are not owned. Rather, ownership passes through a scheduler based on the ownership of the guarantee that the scheduler instance possesses. In other words, if a scheduler receives a guarantee that is owned by user A, then the guarantees that the scheduler provides will also be owned by A. The only exception to this rule occurs when guarantees are delegated from the system to another resource principal.

- **Processor allocation rules**, like guarantees, are owned by a particular resource principal, and determine how CPU time allocated to that principal is to be sub-allocated to applications that request service.

- **Events** cause rules to be evaluated. Events are named, and rules can be configured to run when certain events are signaled. Rules may signal and be signaled by custom events, or they may simply use the built-in events `NEW_REQUEST` and `REQUEST_ENDING`.

All resource manager entities except events and rules are represented as passive data structures. These data structures are available for inspection (and sometimes modification) by rules, which are represented by executable code that is called by events.

8.3 Guarantee Acquisition

When an application requests a guarantee (or when a guarantee is requested on behalf of an application), the guarantee is used for several distinct purposes. First, the resource manager checks if the request violates any system- or user-specified rules. If no rules are violated, the resource manager interprets the guarantee in order to determine which, if any, schedulers in the hierarchy are capable of granting the guarantee. Finally, the guarantee is passed to individual schedulers that perform schedulability analysis based on the values inside of the guarantee.

In response to a new request, the resource manager performs the following actions:

1. It verifies that the request does not violate any processor allocation rules that the user has set up.

2. It finds a set of *candidate schedulers* that (1) can provide a guarantee of the type that is being requested, and (2) have a guarantee that belongs to the system or to the resource principal making the request. Schedulers *advertise* what kinds of guarantees that they can provide. The advertised guarantees are abstract templates that contain unbound numeric parameters. In general, when the scheduling hierarchy is designed well, there should be only one scheduler that can satisfy a given type of request for each user.

3. If there are several schedulers that can provide the requested guarantee, try to acquire the guarantee from each of them in turn.

If the set of candidate schedulers is empty, the request is rejected. It is possible that the resource manager could attempt to load a new scheduler in the hierarchy. We leave this option to future work, since (1) scheduling hierarchies should be simple, and can probably be constructed in skeletal form ahead of time for most situations, (2) experts requiring scheduling behavior not available from default hierarchies will be able to load schedulers manually, and (3) it is unlikely that a heuristic will be able to build a good scheduling hierarchy. So, we currently require the scheduling hierarchy to be pre-built or to be built manually.

8.4 CPU Allocation Rules

8.4.1 Per-User and System Rules

Guarantees describe a contract between a scheduler and a virtual processor about the ongoing allocation of CPU time. Rules describe constraints on the allocation of CPU time that cannot be expressed easily (or at all) using guarantees.

When a guarantee is requested, either by an application or by a scheduler, which may or may not already have a guarantee, it must be shown that the guarantee, if given, would not violate any rules. The basic result of each rule is to either reject a request for a guarantee, or to not reject it. If no applicable rule rejects a request, then the request is passed to a scheduler in the hierarchy, which performs a schedulability analysis to determine if the guarantee is actually feasible. Rules may have side effects, such as signaling an event (possibly invoking other rules) or causing the revocation or renegotiation of a previously granted guarantee in order to grant a guarantee to a more important application.

System-level rules apply to all requests for guarantees and *per-user rules* apply only to the particular principal that instantiated the rule. Typically, system rules will enforce overall fairness between resource principals and enforce system-wide policies (such as mandating non-`NULL` guarantees for time-sharing schedulers). Per-user rules will determine which applications (or kinds of applications) are chosen to run during overload, and also perform special actions such as creating a new scheduler in order to isolate the CPU usage of an untrusted application.

8.4.2 Rule Implementation

Rules are written in C or C++ and run in the address space of the resource manager. Therefore, they cannot be written by untrusted users who could exploit the lack of type safety in these languages to subvert the resource manager. Rather, we envision a number of predefined, parameterizable rules being available to end-users. Although it would be beneficial to have a safe, domain-specific language to write rules in, the design of such a language is beyond the scope of this work.

Rules are made available to the resource manager through dynamic link libraries (DLLs). The resource manager, then, simply provides numerous hooks for user- and administrator-supplied routines to tell it how to allocate CPU time. Rules have access to reflective information about the current set of pending and granted requests, the current set of resource principals, and the scheduling hierarchy.

Users need not select any special rules to implement the default CPU allocation policies of admission control (which is provided by the schedulability analysis routines in reservation-based schedulers) or best effort (which is provided by non-real-time schedulers that simply accept all requests for service).

8.4.3 Rule Evaluation Order

If the signaling of an event triggers more than one rule, they are evaluated in an arbitrary order. It is therefore necessary for rule authors to ensure that the outcome of rule evaluation is not order-dependent. If there are groups of rules that need to be executed in a particular order, this can be accomplished by having the first be rule be triggered by a built-in event such as `NEW_REQUEST`, and then having that rule raise a custom event that triggers that next rule in the sequence.

```
RULE_STATUS Rule_AdmitMostImportant (struct REQUEST_INFO req_info)
{
        // keep terminating the guarantees of less important victim
        // applications until the new request can be admitted or no
        // more victims can be found

        while ((req_pct (req_info) + total_user_pct (req_info->User)) >
               100) {
                struct GUARANTEE min =
                  FindLeastImportant (req_info->User);
                if (min && min->Value < req_info->Value) {
                        EndGuarantee (min);
                } else {
                        return RULE_REJECT;
                }
        }

        return RULE_PASS;
}
```

Figure 8.1: Pseudocode for a rule admitting the subset of applications that have the highest importance

8.4.4 Example Rules

8.4.4.1 Running the Most Important Applications

The pseudocode in Figure 8.1 illustrates the implementation of a per-user resource allocation policy that attempts to admit the subset of the set of applications that have requested service that the user has deemed most important. It makes use of several library functions such as the `req_pct` function to determine the percentage of the CPU that the new request is asking for and the `EndGuarantee` function to revoke a previously granted guarantee.

This rule would be loaded into the system using a command such as this one, which stipulates that the rule is to be triggered by the custom `NEW_REALTIME_REQUEST` event:

```
LoadNewRule (Rule_AdmitMostImportant, NEW_REALTIME_REQUEST);
```

Instead of ending reservations as it runs across them, a real implementation of this rule would need to find a set of reservations to revoke that provides enough extra capacity to grant the new reservation before actually revoking any reservations. This would prevent the bad result of having the rule revoke some reservations, without being able to begin the new one. This rule implicitly assumes that there is a way to convert each guarantee into a percentage, and that CPU addition is linear—that, for example, the total effective utilization of two reservations with utilization 40% is 80%. It is not the case that utilizations can be added linearly for all schedulers; for example, in some cases a reservation scheduler based on rate-monotonic scheduling would be able to grant two reservations each having utilization 40%, and in other cases (depending on the ratio between the periods of the reservations) it would not. A more sophisticated version of this rule could be

```
RULE_STATUS Rule_LimitUserPct (struct REQUEST_INFO req_info)
{
        if ((req_pct (req_info) + total_user_pct (req_info->User)) >
            (100 / num_users ())) {
                return RULE_REJECT;
        } else {
                return RULE_PASS;
        }
}
```

Figure 8.2: Pseudocode for a rule implementing fairness between users

developed that takes non-linear CPU addition into account. Finally, in practice, this rule would need to be coupled with a rule that, each time an application cancels a guarantee, attempts to give a guarantee to the most important feasible application that has a pending request.

8.4.4.2 Enforcing Fairness Between Users

The pseudocode in Figure 8.2 rule implements a global policy that would be useful on multi-user machines. It limits the CPU allocation of each user to his or her fair share of the total available CPU bandwidth.

In practice, this rule would need to be supplemented with another rule that revokes resources reserved by users whose reserved allocation exceeds their new maximum number of shares when a new user enters the system. This could be implemented by creating a custom `RECALCULATE_SHARES` event that either performs the revocation or notifies the user to reduce her resource allocation. Another refinement would be to penalize users for running threads that have very short periods—these threads pollute the cache for other users, effectively reducing the performance of their applications.

8.4.4.3 Other Rules

The following list describes other rules that would be useful to perform in the resource manager:

- A more sophisticated version of the utility-maximizing rule presented above could be implemented. Instead of running the set of applications whose values are largest, it would the set of applications with the largest total value.
- A rule implementing complex scheduling behaviors such as probabilistic CPU reservations.
- A rule could create a new time-sharing scheduler instance when a user logs in, and make it that user's default scheduler. A second rule would be required to destroy the scheduler instance once the user logs out.
- A rule could be used to multiplex requests for CPU reservations among multiple instances of a uniprocessor reservation scheduler. A refined version of this rule might arrange CPU reservations on multiple processors using a bin-packing heuristic to achieve maximal utilization across all processors. A further refinement would be to attempt to place reservations with

similar periods on the same processor, to reduce the expected number of unnecessary context switches.

- A rule could create a new scheduler to run threads belonging to an untrusted application, preventing it from unfairly using more than its share of the CPU.

8.5 Supporting Unmodified Applications

8.5.1 Determining Application Scheduling Parameters

This section proposes a method for determining the scheduling parameters of multi-threaded multimedia applications. This calibration procedure must be run on each hardware platform each time a new real-time application is installed, although vendor-installed real-time software could be pre-calibrated. It will work under the following set of assumptions:

- Threads belonging to the application have periodic CPU requirements.
- The CPU requirements of threads are predictable.
- The application creates the same set of threads each time it is run (but not necessarily in a deterministic order).
- Threads that share the same start routine have the same CPU requirements. (When an application creates a new thread, it passes the address of a *start routine* to the kernel—this address is the first value loaded into the new thread's program counter.)

The calibration procedure operates as follows. On an idle machine (i.e. no other applications are running, services and daemons disabled) the application is run for a long enough period of time that short-term variations in CPU usage are discovered. Instrumentation built into the hierarchical scheduling infrastructure discovers the period and amount of CPU time used by each thread (threads in a real-time application typically use a timer to schedule themselves with a given period). Threads are identified by their start address, which can be discovered either by adding a hook to the kernel routine that creates threads or by having the calibration software attach itself to the real-time application using debugging hooks. The resource manager keeps a checksum of the application executable file to determine when it changes, requiring recalibration. In addition, the resource manager should monitor the status of the operating system, shared libraries, device drivers, and system hardware—a change in any of these can potentially affect the CPU utilization of real-time applications. Fortunately, most end users seldom upgrade any of these system components.

Once the CPU usage of all threads has been identified, an amount of CPU time to guarantee future instantiations of the application can be determined. A conservative estimate would take the maximum amount of CPU time that the application used during any time period and then overestimate by a small factor. The degree of conservatism for each application should reflect the application's importance and how gracefully its performance degrades when receiving less CPU time than it requires. For example, an application that plays music that has small, deterministic CPU requirements could be assigned a generous reservation, allowing it to miss essentially no deadlines.

Applications that do not meet one of the requirements for this method will have to be scheduled in a different way. A progress-based scheduler or a traditional static-priority scheduler can be used to schedule these applications, although without any hard guarantees. Applications whose source code is available can be modified to be self-calibrate using calibration library routines, or simply to operate in a manner that is more conducive to calibration by the external tool.

8.5.2 Applying Scheduling Parameters to Applications

The resource manager should maintain a persistent store of information about real-time applications. This information can be supplied by users, by system vendors, or by the calibration procedure described in the previous section. It may be tuned by sophisticated users from time to time to reflect changing preferences about the degree of pessimism different applications require.

When an application begins execution, it can be identified either by name or (more securely and accurately) by a strong checksum of its executable file. Once an application is identified as being one that should receive real-time scheduling, the resource manager can internally request a guarantee for the application.

8.6 Implementation

The resource manager is invoked infrequently compared to the number of low-level scheduling decisions—typically only when an application requests real-time service, has a change of requirements, or exits. So, the resource manager can be implemented as a user-level process, with requests being performed though remote procedure calls generated by a library. It interacts with the scheduling hierarchy in the kernel through the `HLSCtl` interface, to which it has exclusive access. Since the scheduler infrastructure assigns new threads to a default best-effort scheduler, only applications that request other types of scheduling behavior need to interact with the resource manager.

8.7 Doing Without the Resource Manager

The resource manager is designed to provide a flexible level of indirection between requests for scheduling and the scheduling hierarchy. It also allows high-level rules about processor allocation to be enforced.

In a system without a resource manager, the only supported resource management policies are those implemented by the schedulers themselves. For example, admission control for reservation-based schedulers and best-effort for time-sharing and proportional share schedulers that do not perform reservation. Thus, the burden of running a mix of applications that provides high value falls on the user.

Without the resource manager, special-purpose code libraries and scripts can be written to help manage the scheduling hierarchy. These programs are particularly easy to write if they can make assumptions about the structure of the scheduling hierarchy rather than attempting to dynamically find a scheduler that can meet an arbitrary request. For example, the `hlsutil` command that was implemented for the HLS prototype allows a CPU reservation to be started or ended for an arbitrary thread. It does this by passing the name of the thread and the parameters of its potential reservation to scheduler infrastructure using the `HLSCtl` command. The infrastructure then locates the reservation scheduler by name, and requests real-time scheduling for the thread. `hlsutil` can manipulate the scheduling hierarchy in a number of other ways. For example, it can create and delete a scheduler, change the default scheduler, move a thread between schedulers, move all threads from the default scheduler to a different scheduler, and change the clock interrupt frequency.

8.8 Conclusions

This section has presented the design of the HLS resource manager, a middleware application that provides a level of indirection between applications and the scheduling hierarchy, allowing high-level policies about processor allocation to be enforced. The resource manager can add value, but the utility of the scheduling hierarchy is not contingent on having a resource manager.

Chapter 9

Implementation of HLS in a General-Purpose OS

This chapter describes the implementation of the hierarchical scheduler infrastructure in the Windows 2000 kernel. Also, the implementation of four loadable schedulers is described: a time-sharing scheduler that can also act as a fixed-priority scheduler, a proportional share scheduler, a reservation scheduler, and a join scheduler.

9.1 Background and Implementation-Level Design Decisions

9.1.1 Windows 2000 Background

It is necessary to present some background about the structure of Windows 2000 before going into detail about the execution environment for loadable schedulers.[1] Windows 2000, like its predecessor Windows NT, is a preemptible multiprocessor operating system. This means that multiple application threads can be executing in the kernel concurrently. Unlike traditional Unix-like operating systems, threads executing in the kernel can be preempted by other threads. Therefore, *thread dispatch latency*—the time between when a thread becomes ready and when it first begins to execute—is usually very low for high-priority threads. In contrast, in a non-preemptible operating system a high-priority thread may awaken while a low-priority thread is executing a slow system call. This leads to a priority inversion (a low-priority task preventing a high-priority task from running) until the system call finishes.

Each processor in Windows 2000 has an associated *interrupt request level* (IRQL) that determines what events may preempt the activity running on that processor. Normal user- and kernel-level threads (of any priority) run at *passive IRQL*; they may be preempted by any hardware or software interrupt handler. Each hardware interrupt handler executes at a specific IRQL. To protect against being preempted by an interrupt, it is sufficient for a block of code to raise the processor IRQL to the IRQL of the corresponding interrupt. This is because an interrupt is signaled on a processor only when the processor's IRQL is lower than the IRQL of the interrupt.

The Windows 2000 scheduler executes at *dispatch IRQL*, in the context of a software interrupt called the *dispatch interrupt*. All interrupt service routines for hardware devices have higher IRQLs than the dispatch IRQL, which is higher than the passive IRQL. Typical use of the dispatch interrupt

[1] Chapters 3 and 6 of Solomon and Russinovich [77] describe Windows 2000 dispatching and scheduling mechanisms in considerably more detail.

would be as follows: a keyboard interrupt handler awakens a thread that was blocked waiting for keyboard input. As part of the process of awakening the thread, the kernel requests a dispatch interrupt. Because the IRQL of the keyboard interrupt handler is higher than the dispatch IRQL, the dispatch interrupt is deferred until the keyboard interrupt handler exits, restoring the processor IRQL to its previous value. As soon as the IRQL is below the dispatch IRQL, the dispatch interrupt is signaled and the Windows 2000 scheduler potentially dispatches the newly awakened thread.

In addition to running scheduler code, the dispatch interrupt handler has another function: running *deferred procedure calls* (DPCs). DPCs were designed to give device drivers access to high-priority asynchronous processing outside of the context of hardware interrupt handlers. Because code running at dispatch IRQL or higher cannot be preempted by the scheduler, it is essential that code spend as little time as possible running with elevated IRQL.

Mutual exclusion in Windows 2000 is provided by both *blocking locks* and *spinlocks*. Blocking locks are suitable for protecting resources that may be busy for a long period of time. Code that executes at dispatch IRQL or higher may not block: this leads to deadlock since the scheduler, which needs to select the next thread, cannot be entered until the IRQL falls below the dispatch IRQL. Spinlocks always run at dispatch IRQL or higher.[2]

All Windows 2000 scheduler data structures are protected by a spinlock called the *dispatcher database lock*. Therefore, access to the scheduler is serialized across all processors. Contention for the dispatcher database lock is a potential cause of scalability problems on large multiprocessors that run scheduler-intensive workloads—other multiprocessor operating systems such as IRIX and Solaris have per-processor scheduler data structures, requiring less synchronization between processors.

9.1.2 The Loadable Scheduler Execution Environment

Hierarchical schedulers are implemented as loadable device drivers. In Windows 2000, a subset of the internal kernel APIs are available to loadable drivers, but by default this subset is not powerful enough to allow drivers to act as schedulers. The hierarchical scheduler infrastructure exports the additional kernel entry points that are necessary for loadable modules to act as schedulers.

9.1.2.1 Serialization

The scheduling hierarchy, like the native Windows 2000 scheduler, is protected by the dispatcher database lock. Thus, the scalability of HLS is only worse than the scalability of the native Windows 2000 scheduler to the extent that hierarchical schedulers are less efficient. Removing the dispatcher database lock in order to provide per-processor scheduling data structures in Windows 2000 would be a major undertaking, and is beyond the scope of this work.

9.1.2.2 Blocking

Since it executes at dispatch IRQL, loadable scheduler code must not incur a page fault or block for any other reason. In general, it does not make sense for a scheduler to block, since schedulers do

[2]Spinlocks must run at at least the IRQL of the highest-IRQL interrupt that may acquire the spinlock. This is to prevent code holding a spinlock from being preempted by an interrupt handler that will acquire the same spinlock—this would deadlock the processor.

Windows 2000 state	HLS state
Initialized Terminated	*no equivalent*
Waiting Transition Ready (on process ready queue)	Waiting
Ready (not on process ready queue)	Ready
Standby Running	Running

Table 9.1: Mapping of Windows 2000 thread states to HLS virtual processor states

not encapsulate a control flow in the same way that threads do. Rather, schedulers are coroutine-based entities that execute in a restricted environment. When a user-level thread sends a message to a scheduler, the scheduler infrastructure helps schedulers avoid blocking by copying data from the process address space (potentially incurring a page fault) into non-pageable memory while still executing in the context of the user thread. Then, the dispatcher database lock is acquired and schedulers may access both their own data structures and data copied from the requesting thread without risk of blocking.

9.1.2.3 Memory Allocation

To avoid blocking, all memory that is reachable from a loadable scheduler must be non-pageable (that is, pinned to physical memory pages). Unfortunately, Windows 2000 makes it impossible to dynamically allocate non-pageable memory while the kernel is executing at dispatch IRQL because the memory allocation routine itself may block when there is a shortage of free memory pages. HLS works around this difficulty by allocating a block of memory at boot time and sub-allocating it using its own versions of `malloc()` and `free()`. Schedulers are assumed to not require more dynamic memory than HLS reserved at boot time.

9.2 Implementation of the Hierarchical Scheduler Infrastructure

At a high level, to ensure proper operation of loadable schedulers it is necessary for the HSI (1) to be notified of each relevant OS event such as thread blocking, unblocking, creation, and deletion, (2) to be able to make scheduling decisions (i.e., to dispatch threads), and (3) to defeat the native Windows 2000 scheduler, preventing it from ever making a scheduling decision.

9.2.1 Simplifying Thread States

There are seven possible thread states in Windows 2000, in addition to an anomalous condition where a thread is in the *ready* state but is not able to run. These states can be divided into two groups: states that concern the CPU scheduler, and states that serve some other purpose. HLS only allows hierarchical schedulers to see the states that could affect a thread scheduler. Table 9.1 shows the mapping from Windows 2000 states to HLS states. Transitions between Windows 2000 states that HLS treats as equivalent do not result in notifications being sent to the scheduler infrastructure. Transitions between states that are not equivalent must result in notifications being sent.

Initialized threads have some of their data structures allocated but are not yet ready to run. Similarly, *terminated* threads will execute no more instructions, but have not yet been deallocated. The HSI is designed such that hierarchical schedulers never see threads in either of these two states.

The *waiting* state has the same meaning in Windows 2000 as it does in HLS: threads in that state cannot execute until some external condition is satisfied. When Windows 2000 is under memory pressure, it can reclaim memory in a number of ways. One of them is to swap out the kernel stacks of blocked threads, and another is to swap out entire processes; this can be done when all threads belonging to a process are blocked. Threads whose kernel stacks have been swapped out are put into the *transition* state while they wait for the stack to be paged back into memory. This special case of the waiting state is irrelevant to HLS and it considers threads to be waiting while they are in transition.

Similarly, when a thread is awakened and its process is swapped out, the thread moves into the ready state and it is marked as being on the *process ready queue*. This condition is misleading as the thread in not actually ready to execute: it must wait for its process to be swapped back into memory before it is taken off of the process ready queue and made runnable. In other words, a ready thread that is on the process ready queue is, for all practical purposes, waiting—it is treated as such by HLS.

The *running* state has the same meaning in Windows 2000 and HLS: it means that a thread is executing on a physical processor. The Windows 2000 state *standby* denotes a thread that is "on deck," or about to execute. The standby state is used by Windows 2000 as follows. Assume that a high-priority thread running on processor 0 exits a critical section, allowing a low-priority thread to wake up and enter the protected region. The low-priority thread cannot preempt the high priority thread, but it can be dispatched on a different processor. In this case, Windows 2000 marks the low-priority thread as being on standby on another processor and sends the processor an interprocessor interrupt. When the IPI is received, the other processor enters low-level dispatching code that sees the standby thread and dispatches it. HLS insulates schedulers from low-level details like this, and considers a thread to be running as soon as it enters the Windows 2000 standby state.

9.2.2 Suppressing the Native Windows 2000 Scheduler

Although schedulers in the hierarchical scheduler architecture subsume the functionality of the native Windows 2000 time-sharing scheduler, the Windows 2000 scheduling code still executes in an HLS system—it was not eliminated since the prototype HLS implementation attempts to be as unintrusive as possible. Since the Windows 2000 scheduler is never permitted to make a scheduling decision it is mostly vestigial, but it still performs a few useful functions. Windows 2000 contains a number of specialized scheduling heuristics such as raising the priority of threads after they unblock, boosting the priorities of threads that have been runnable but not scheduled for several seconds, and extending the length of scheduling quanta of threads in the foreground process. The latter heuristic is not carried over from the native Windows 2000 scheduler to HLS (although it could be implemented separately in a hierarchical scheduler). The first two heuristics, however, use an internal kernel interface to raise the priority of a thread. Their benefits are preserved in HLS by having bottom schedulers convert a request to set thread priority into an HLS message and sending the message to its parent. If the parent scheduler is not capable of acting on this message (for example, if it is a reservation scheduler), the message is ignored. Otherwise, the priority boost and subsequent decay happen in HLS just as they did under the native Windows 2000 scheduler. The end result of this is that the default HLS scheduler, a priority and round-robin scheduler described in Section 9.3.1, behaves very much like the native Windows 2000 scheduler.

9.2.3 Allowing the HSI to Schedule Threads

When a hierarchical scheduler grants a processor to a bottom scheduler, the thread that the bottom scheduler represents can usually be immediately dispatched. To do this, the scheduler sets the thread's state to standby on the processor that was granted and then requests a dispatch interrupt for that processor. The dispatch interrupt handler then moves the thread from standby to running and switches to that thread's context.

This simple method for scheduling threads does not work when the thread to be scheduled is already running on a processor other than its intended target. In this case, the HSI requests dispatch interrupts on both the current and target processors. Since the dispatch interrupt handler acquires the dispatcher database lock, the handlers are serialized. If the processor that is currently running the thread runs first, then it can deschedule the thread, allowing the other processor to schedule the thread normally. However, there is a complication if the target processor enters the dispatch interrupt handler first—it cannot schedule the thread until the other processor has preempted it and saved its state. To deal with this case, the HSI forces the target processor to idle until the thread's current processor moves the thread from the running to the ready state, at which point the target processor can put the thread back into the running state and dispatch it.

The reality is considerably more complicated than this two-case analysis, since in between a processor idling and picking up the thread, a scheduler could decide to schedule that thread on a completely different processor, the thread could exit, it could be requested to be moved to a different scheduler, etc. The HSI hides this complexity from loadable schedulers.

9.2.4 Interposing on Windows 2000 Scheduling Events

Since Windows 2000 was not designed to support loadable schedulers, suppressing its scheduling decisions and implementing new scheduling behavior required inspecting several dozen kernel source files that have to do with thread management, and changing some of them. Table 9.2 shows the number of non-trivial modifications to Windows 2000 that were required to notify the HSI of all events of interest, allow it to make scheduling decisions, and to suppress the native scheduler.

Early in the loadable scheduler implementation, a passive version of these changes was implemented, whose purpose was to "shadow" the state changes of Windows 2000 threads. In other words, the HSI never made a scheduling decision, but rather, it kept an up-to-date version of the state of each thread in the system, and signaled an error if this state ever disagreed with the state that Windows 2000 believed the thread to be in. After thoroughly inspecting the native Windows 2000 scheduling code and subjecting a system containing a passive version of the HSI to memory pressure and high load without observing any disagreements in thread states, it was assumed that these changes correctly notify the HSI of all events of interest.

The two modifications listed as "other" in Table 9.2 refer to extensive modifications to the scheduler routine `KiReadyThread` and to the idle loop.

Functionality	Modified code sites
Initialize HSI	1
Notify HSI of blocking thread	7
Notify HSI of unblocking thread	8
Notify HSI of thread create	3
Notify HSI of thread exit	1
Notify HSI of change in thread priority	4
Suppress native Windows 2000 scheduler	11
Other	2

Table 9.2: Summary of changes to Windows 2000 to support hierarchical scheduling

9.2.5 Time and Timers for Loadable Schedulers

Schedulers need to be able to accurately determine the current time, and also to arrange to regain control of the CPU at a specific time in the future.

Reading the current time is cheap and accurate on Pentium-class x86 processors; the `rdtsc` instruction returns the number of cycles since the machine was turned on.[3] One division and one addition are required to turn this into a Windows 2000 filesystem time (64-bit integers in 100 ns units representing either a time relative to the present or relative to Jan 1, 1601).

The HSI uses *Windows kernel timers* to get control of a CPU at a particular time. Kernel timers call a user-supplied routine in the context of a DPC. DPCs, like other scheduler operations, run at dispatch IRQL. When the HSI receives a timer callback, it acquires the dispatcher database lock before calling the `TimerCallback` function of the scheduler whose timer expired. Instead of using a single kernel timer and dispatching expirations to the correct scheduler, the HSI simply allocates a separate kernel timer for each loadable scheduler instance.

The precision of kernel timers is limited by the granularity of the clock interrupts on a particular system. By default, clock interrupts arrive every 10–15 ms, depending on which *hardware abstraction layer* (HAL) is being used. This granularity is too coarse for some multimedia applications. On some HALs, Windows 2000 allows the timer interrupt frequency to be changed at run-time. The HSI arranges at boot time for the timer frequency to be 1024 Hz, the highest frequency supported by the released Windows 2000 distribution. By modifying the HAL used for multiprocessor PCs, HLS is able to drive the *real-time clock* (RTC)—hardware device that this HAL uses to generate clock interrupts—at up to 8192 Hz. For the work reported in Chapter 11 it was run at 4096 Hz in order to make timer expirations precise to within about 0.25 ms. This is a fair approximation of a precisely settable timer—a feature that would be found in any OS specifically designed to support dynamic real-time scheduling.

The presence of a periodic clock interrupt instead of a precisely settable timer in Windows 2000 is typical of time-sharing operating systems. Periodic clocks are used in these OSs because (1) GPOSs were not designed to support real-time applications requiring precise timing facilities,

[3] The usefulness of the cycle counter will decrease as it becomes common for microprocessors to dynamically alter their clock speed as part of power and heat management strategies. Of course, real-time scheduling will also be difficult on these processors.

and (2) limiting the clock granularity leads to the batching up of timed events, which tends to increase system throughput by limiting the number of times the running application is preempted and has the cache polluted.

9.3 Schedulers Implemented

Schedulers implemented for HLS include a time-sharing scheduler, a fixed-priority scheduler, a proportional share scheduler, and a real-time scheduler that provides basic, hard CPU reservations. Each of these schedulers was chosen because it is representative of a broad class of schedulers that have been used to schedule multimedia applications. Using these schedulers as basic building blocks, complex scheduling behaviors can be built, as Section 6.2 and Chapter 7 showed.

9.3.1 Time Sharing

The HLS time-sharing scheduler is a multiprocessor priority and round-robin based scheduler that is very much like the native Windows 2000 scheduler. It has several capabilities that the native scheduler does not, however. First, it is a hierarchical scheduler, meaning that it correctly handles processor revocation. Second, it can be configured to schedule any number of processors, not just the number of processors that are physically present on the system. And third, in addition to being a round-robin scheduler it can be configured to act as a non-preemptive fixed-priority scheduler.

There are also several features not supported by the HLS time-sharing scheduler that are supported by the native Windows 2000 scheduler. First, the HLS time-sharing scheduler does not support variable sized scheduling quanta, except at compile time. Second, it makes no attempt implement *processor affinity*, to keep threads running on the processor that they had previously been running on. Third, it does not support *affinity masks* that restrict a thread, or all threads in a process, to run on a subset of the full set of processors. Support for affinity masks could be added to the HLS time-sharing scheduler in a straightforward way, but we did not do this because standard time-sharing and multimedia applications do not appear to make use of them.

A final difference between the two schedulers is that the HLS time-sharing scheduler maintains the invariant that a thread will never be in the ready state while a lower priority thread is in the running state. On a multiprocessor machine the native Windows 2000 scheduler does not make this guarantee: to a certain extent it elevates processor affinity above the requirement to always run the set of tasks that have the highest priorities.

9.3.2 Rez

Rez is a scheduler that provides basic, hard CPU reservations. The algorithm it uses is similar to a number of other reservation schedulers that have been described in the literature such as the *constant utilization server* developed by Deng et al. [18], the *constant bandwidth server* that Abeni and Buttazzo developed [2], and the Atropos scheduler developed for the Nemesis OS [49].

Rez assigns a budget to each thread that has a CPU reservation. Budgets are decremented in proportion to the CPU time allocated to the associated thread, and are replenished at the beginning of each period. Rez always schedules the thread that has the earliest deadline among all threads that are runnable and have a positive budget. The deadline for each thread is always taken to be the end of its current period.

Rez is a uniprocessor scheduler in the sense that it can only schedule one CPU worth of reservations, but it can be run on multiprocessors by instantiating an instance for each processor.

9.3.3 Proportional Share

The hierarchical proportional share (PS) scheduler implements the *start time fair queuing* (SFQ) algorithm, with a *warp* extension similar to the one implemented in BVT [20]. However, the PS scheduler does not provide support for the *warp time limit* and *unwarp time requirement* that BVT supported. Each thread is assigned a warp value (defaulting to zero) that allows threads to borrow against their future processor allocation, providing a mechanism for low dispatch latency for time-sensitive threads.

The PS scheduler can also be compiled without support for warp, making it amenable to the throughput and latency bounds for SFQ that were described by Goyal et al. [28]. Each SFQ thread has an associated *virtual time* that increases in proportion to the time the thread runs and in inverse proportion to the thread's *share*. Threads that block, upon awakening, are forced to "catch up" with the virtual time of the currently running thread, meaning that blocked threads do not build up credit. The PS scheduler always dispatches the thread with the smallest virtual time among all runnable threads.

The proportional share scheduler currently interprets thread priorities as proportions. In other words, an application running at the default priority (8) would receive eight times the amount of processor time received by a thread running at the idle priority (1). This was an expedient way to make use of the existing infrastructure for manipulating priorities, and to provide applications that are not aware of the PS scheduler with scheduling behavior that approximates what they would have received under the native Windows 2000 scheduler.

9.3.4 Join

The function of most schedulers is to multiplex a physical processor, or part of a physical processor, among many virtual processors. *Join* schedulers have the opposite purpose: to schedule a single virtual processor whenever any one of its parents schedules it.

On a multiprocessor, there is the possibility for multiple parents to grant physical processors to a join scheduler at the same time. To handle this case, the join scheduler must have a policy for deciding which processor to use. A simple priority scheme is likely to be effective in most cases, and is the policy implemented in the HLS join scheduler. To implement the priority scheme, the join scheduler numbers its parents and always accepts scheduling from the highest-numbered parent that will grant a processor to it.

9.3.5 Top

The top scheduler allows only as many virtual processors to register for scheduling as there are physical processors in the system. It always immediately grants each request made to it, and furthermore, always grants the same physical processor to each virtual processor (unless the VP unregisters and re-registers).

9.3.6 Bottom

The bottom scheduler is responsible for actually dispatching threads and for converting thread events into HLS notifications. When a thread event occurs, it is desirable to quickly find the associated bottom scheduler instance. To this end a pointer to the bottom scheduler instance was added to the `ETHREAD` structure (the "executive thread block" that Solomon and Russinovich [77, pp. 317–327] discuss). Adding fields to kernel internal data structures in Windows 2000 is risky because some structures are referenced from assembly language routines that use hard-coded offsets to locate particular structure elements. However, adding a pointer to the end of the executive thread block did not appear to cause any problems. The executive thread block is the only Windows 2000 kernel data structure that was modified while implementing HLS.

9.4 HLS Implementation Experience

9.4.1 A Simulation Environment for HLS

The hierarchical schedulers and infrastructure can be compiled to run either in the Windows 2000 kernel or in a user-level application as part of an event-driven simulation. The simulation environment models just enough of the Windows 2000 kernel environment for the schedulers to run. It models threads as programmable state machines, allowing them to perform arbitrary scheduling actions while the simulation is running. For a number of reasons, the simulation was an invaluable tool during the development of HLS.

- The simulation enabled the testing of unavailable hardware configurations (a 17-processor machine, for example).
- Tools such as a graphical profiler, a graphical debugger, and Purify (a run-time tool that detects memory leaks, array-bound violations, and many other classes of dynamic errors that are common in C programs) could be used to help debug the simulated version of HLS.
- A different compiler, gcc, could be used to compile the simulated version of HLS. This helped catch questionable constructs since gcc produces different warnings than the Microsoft compiler.
- Since the simulator is deterministic, the same scenario could be run repeatedly, facilitating debugging.
- The simulator took the time to boot Windows 2000 out of the debug cycle, enabling much faster testing after making a change to HLS.
- All bugs discovered in the simulation were guaranteed to be (1) sequential bugs instead of race conditions, and (2) bugs in high-level parts of HLS rather than in the Windows 2000 interface code (which is not compiled into the simulation). These two extra bits of knowledge made some bugs much easier to track down.
- Brute-force randomized testing could be performed in the simulation. For example, it was possible to set up a simulated machine with a random number of processors and a random number of threads with random behavior (beginning and ending reservations, blocking and unblocking, etc.).

- Results of simulation runs could be visualized easily because the simulator dumps an event log to a file that can be quickly converted to a graphical execution trace by a Perl script and a plotting utility.

The disadvantage of the simulation environment was the time it took to implement it. However, this was far outweighed by the advantages. In particular, randomized testing appeared to be very good at finding corner-cases that were difficult to think of while implementing the code—the amount of state in a hierarchy containing several schedulers plus the machine state and HSI state made the behavior of the entire system too complicated to easily reason about. In general, either randomized testing found a bug within a few minutes, or failed to find any bugs at all (which meant not that HLS was bug free, but that it happened to work under a particular simulated workload).

9.4.2 Code Size Results

In addition to the modifications to Windows 2000 listed in Table 9.2, implementing the HSI required adding about 3100 lines of code to the Windows 2000 kernel. The line counts for the schedulers are as follows: 1176 for the time-sharing scheduler, 1058 for Rez, 765 for the proportional share scheduler, 477 for the join scheduler, 218 for the top scheduler, and 392 for the bottom scheduler. These line counts include a substantial amount of comments and debugging code. For example, comment lines, blank lines, and debugging code accounted for 413 lines of the proportional share scheduler; after removing them only 352 lines remained.

9.4.3 Developer Effort Results

This section presents some anecdotal evidence about the effort required to implement various loadable schedulers. The top and bottom schedulers are part of the scheduler infrastructure, and the time-sharing scheduler was debugged concurrently with the HSI, so the time taken to develop these is not relevant. It took about a day of coding to write a version of Rez that provided one CPU reservation, and another day to extend it with EDF scheduling to handle an arbitrary number of reservations. Given Rez as a working model of a time-based, root-only uniprocessor scheduler, implementing a root-only version of the proportional share scheduler took only a few hours; a few more hours were required to extend it to correctly handle processor revocation. Writing the join scheduler, in addition to some glue code that provides soft CPU reservations using the join scheduler in conjunction with Rez, took about five hours. Finally, the simulation environment was nearly always sufficient to debug loadable schedulers—additional problems were only rarely uncovered while running them on a real machine.

9.4.4 Lessons Learned

The feasibility of implementing the HSI in operating systems other than Windows 2000 has not been investigated in detail. However, given the similarities among thread abstractions and time-sharing schedulers in general-purpose operating systems, it seems likely that the hierarchical scheduler architecture could have been developed in a multiprocessor operating system such as Linux or FreeBSD with about the same effort as it took to implement it in Windows 2000. Furthermore, since the HSI and loadable schedulers are now debugged and known to work, the HSI could most likely be ported to one of these operating systems relatively easily. Porting the HSI to an OS such as

Solaris or IRIX that supports per-processor scheduling queues may be considerably more difficult, since the scheduling model in these OSs does not closely correspond to the HLS model.

The scheduling structure of a typical general-purpose OS is a cross-cutting *aspect*—a feature of the code that does not happen to be cleanly separated from unrelated code because it was not designed to be changed [43]. Implementing HLS can be viewed as untangling the scheduling aspect from other low-level kernel code.

9.5 Conclusion

This chapter has described the implementation of the hierarchical scheduler infrastructure and several hierarchical schedulers in Windows 2000. The important property of the HSI is that it makes it easier to implement new schedulers by exposing useful scheduling primitives while hiding irrelevant details about the operating system. The schedulers that have been implemented for HLS—a fixed priority scheduler that also serves as a time-sharing scheduler, a proportional share scheduler, a reservation scheduler, and a join scheduler—are representative of the kinds of schedulers typically used to schedule time-sharing and soft real-time applications on general-purpose operating systems.

Chapter 10

Performance Evaluation of HLS

Chapters 4 and 9 established the overall feasibility of HLS in a general-purpose operating system by describing its design and a prototype implementation. This chapter provides additional support for the feasibility of HLS by showing that the run-time cost of flexible scheduling is modest. This chapter also supports the usefulness of HLS by providing quantitative data about two scheduling hierarchies whose scheduling behavior cannot be achieved using a traditional time-sharing scheduler.

10.1 Test Environment and Methodology

All performance tests were run on a dual Pentium III 500 MHz that, unless otherwise stated, was booted in uniprocessor mode. The test machine had 256 MB of memory and was equipped with an Intel EEPro 10/100 network interface and a 36 GB ultra fast wide SCSI disk.

The duration of brief events was measured using the Pentium timestamp counter by executing the `rdtsc` instruction. This instruction returns a 64-bit quantity containing the number of cycles since the processor was turned on. On a 500 MHz processor, the timestamp counter has a resolution of 2 ns.

All confidence intervals in this chapter were calculated at 95%.

10.2 Cost of Basic HLS Operations

Table 10.1 shows times taken to perform a variety of HLS operations from user level. That is, the total elapsed time to perform the given operation including: any necessary user-space setup, trapping to the kernel, checking parameters and copying them into the kernel, performing the operation, and returning to user level. These costs are incurred during mode changes: when an application starts, ends, or changes requirements, rather than being part of ordinary scheduling decisions. Nevertheless, besides loading and unloading a scheduler HLS operations are cheap: they all take less than 40 μs on a 500 MHz CPU.

Except for loading and unloading a scheduler, which uses native Windows 2000 functionality, all HLS services are accessed through the `HLSCtl` system call. Due to cache effects and dynamic binding of the dynamic link library (DLL) containing kernel entry points, the first few times each operation was performed took highly variable amounts of time. To avoid measuring these effects,

Operation	μs per op.
load scheduler	59700.0±553.0
unload scheduler	57600.0± 53.0
create scheduler instance	25.0± 1.63
destroy scheduler instance	18.0± 0.0518
move thread between schedulers	13.3± 0.0215
begin CPU reservation (same process)	15.4± 0.0311
end CPU reservation (same process)	13.5± 0.0272
begin CPU reservation (different process)	36.4± 0.0376
end CPU reservation (different process)	35.0± 0.0649

Table 10.1: Time taken to perform representative HLS operations from user level

each operation was performed 60 times and the first 10 results were thrown away, leaving 50 data points.

The costs of loading and unloading a scheduler, respectively about 60 ms and 58 ms, were calculated by measuring the time taken by a C program to run the commands `system (''net start driver'')` and `system (''net stop driver'')`. The `net` command is used to manually load device drivers in Windows systems. The time taken by this command includes considerable overhead not directly related to loading the driver: for example, starting a new command shell and invoking the `net` command. To avoid the possibility of having this time in the critical path of an application requesting real-time scheduling, scheduler device drivers could be arranged to be loaded into the kernel at boot time, or schedulers can simply be built into the kernel binary.

Creating a scheduler instance requires the following actions to be performed: allocating a block of memory for per-instance data, adding the scheduler to a global list of scheduler instances, and initializing data structures used to provide a timer to the scheduler. The new instance is then ready to be used, but it is not part of the scheduling hierarchy until it registers one or more virtual processors with another scheduler. Destroying a scheduler involves removing it from the global list of scheduler instances and deallocating its data block.

All threads, when created, belong to a default time-sharing scheduler. They may move, or be moved, to a different scheduler at a later time in order to receive a different scheduling behavior. Moving a thread involves releasing the processor if the thread is in the ready or running states, unregistering the thread's bottom scheduler's virtual processor from the current scheduler, registering the virtual processor with the new scheduler, and then requesting scheduling if the thread was previously in the ready or running state.

Beginning and ending a CPU reservation is equivalent to moving a thread from its current scheduler to a reservation scheduler and then requesting a CPU reservation. Ending a reservation moves the thread back to the original scheduler. There are two cases for beginning and ending a CPU reservation. In the first case, the thread that is requesting a CPU reservation belongs to the same process as the thread that will receive the CPU reservation. In the second case, the two threads belong to different processes. The first case is faster since within a process, a *handle* to the target thread will be known. Handles are reference-counted indirect pointers to Windows 2000 kernel objects; their scope is a single process. When a request is made to begin or end a CPU reservation

Scheduler	Median context switch time (μs)
Released Windows 2000	7.10
Rebuilt Windows 2000	7.35
HLS time sharing	11.7
HLS CPU reservation	12.5
HLS proportional share	19.9

Table 10.2: Context switch times for HLS and native Windows 2000 schedulers

for a thread in a different process, a handle must be acquired using the `OpenThread` call before the request is made, and this takes time. `OpenThread` returns a handle when given a *thread ID*, a global name for a thread.

10.3 Context Switch Times

The previous section described HLS operations that are not performed often, compared to the frequency of individual scheduling decisions. Since scheduling decisions are implicit, or not directly requested by threads, it is more difficult to accurately measure their costs compared to the operations in the previous section. The strategy taken in this section is to measure the effect of HLS on *context switch time*—the time between when one thread stops running and another thread starts. Context switches occur when a thread blocks and, potentially, when a thread unblocks. They also occur at scheduler-determined times: for example, a quantum-based scheduler such as a proportional share scheduler or a traditional time-sharing scheduler preempts a thread when its quantum expires in order to run a different thread.

10.3.1 A Single Loadable Scheduler vs. Unmodified Windows 2000

Table 10.2 shows the effect that HLS has on thread context switch times. These numbers were calculated by running a test application that created ten threads and let them run for 100 seconds. Each of the threads continuously polled the Pentium timestamp counter in order to determine when the thread was running and when it was not. By collating the execution traces of all ten threads it was possible to find the durations of the time intervals between when one thread stopped running and when another started. This time, which was not available to any thread because it was spent in the kernel, is the context switch time. The scheduling hierarchy used to generate these figures had two levels: a fixed-priority scheduler at the root of the scheduling hierarchy that scheduled a reservation scheduler at high priority and a time-sharing scheduler at low priority.

Data collected in this manner is "polluted" by other events such as the execution of system threads, interrupt handlers, and DPCs. These events artificially inflate some measurements of context switch time, meaning that a histogram of context switch times will have a long tail. Figure 10.1 shows such a histogram. The smallest context switch time in this data set was 5.54μs, the median value was 7.10μs, and the maximum value was 1740μs—in other words, the tail of the histogram continues far past the right-hand border of the figure. The high maximum value for a context switch does not mean that any context switch actually took that long, but rather that some thread other

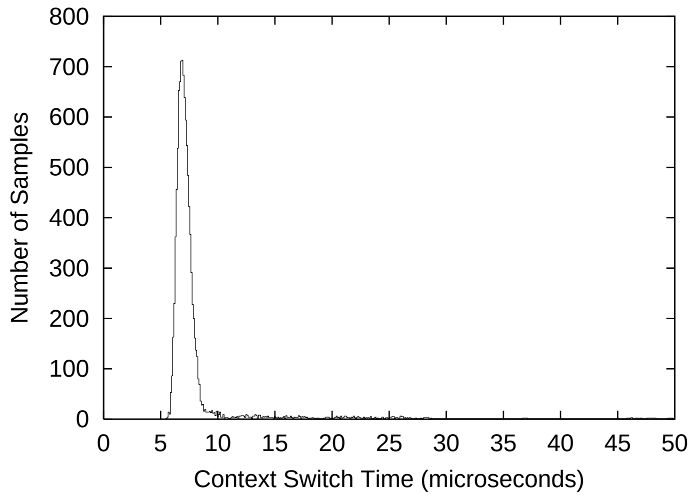

Figure 10.1: Histogram of context switch times for the released Windows 2000 kernel

than one belonging to the context switch test application ran for about 1.7 ms at some point during the test. Since the averages of distributions with long tails are not meaningful, we take the context switch time to be the median of the set of measured times.

The *released* version of Windows 2000 is the kernel binary that would be installed by an off-the-shelf copy of Windows 2000. The *rebuilt* version of Windows 2000 is a kernel compiled from unmodified sources. The context switch time for the released kernel is slightly faster than the context switch time for the rebuilt kernel because post-compilation optimizations are applied to released Windows 2000 kernels—these are not available as part of the Windows source kit.

For the experiment used to generate each median context switch time reported in Table 10.2, all threads belonging to the test application were scheduled by the same scheduler. This indicates that context switches within a well-written HLS scheduler are about 60-70% more expensive than context switches in the rebuilt version of Windows 2000. The proportional share scheduler cannot be considered to be well written: it is considerably less efficient than the other schedulers since it uses a naive linear search (among all threads, not just runnable ones) when searching for a thread to dispatch. The performance of this scheduler could be improved by scanning only runnable threads and/or using a sub-linear search algorithm.

The impact of increased context switch time on overall application throughput depends heavily on how many context switches applications incur. For example, multimedia applications that on average run for 1 ms before being context switched incur 0.71% overhead from the native Windows 2000 scheduler, and 1.25% overhead from the HLS reservation scheduler. This overhead is likely to be acceptable for home and office machines where high throughput is not critical; it may not be acceptable for server machines. However, there are a number of optimizations that could be applied to HLS in order to improve context switch time—these are discussed in Section 10.6. Furthermore, in Section 10.3.3 we will show that the true cost of context switches, including the time lost to applications as they rebuild their working sets in the cache, can be much higher than the numbers presented in this section.

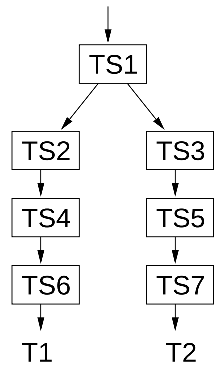

Figure 10.2: Scheduling hierarchy used to measure the cost of a context switch involving 4 levels of schedulers

10.3.2 Effect of Hierarchy Depth on Context Switches

The previous section measured the cost of context switches involving a scheduling decision made by a single HLS scheduler. This section measures the effect of varying the number of schedulers involved in a scheduling decision. Figure 10.2 shows the scheduling hierarchy that was used to measure the context switch time of a 4-level scheduler. Assume that TS1 has allocated the CPU to TS2, allowing T1 to run. At the end of TS2's time slice, TS1 revokes the processor from TS2; the revocation proceeds down the chain until it reaches T1, whose bottom scheduler releases the processor. Once the release notifications propagate back to TS1, it grants the processor to TS3; once the grant notifications propagate to T2, a scheduling decision has been made. Analogous hierarchies were used to measure the cost of context switches for hierarchies of other depths.

Figure 10.3 shows how context switch time varies with hierarchy depth. The increase in cost is nearly linear. The slope of the line of best fit is 0.961, meaning that each additional level of the scheduling hierarchy adds approximately 961 ns to the context switch time. In practice it is unlikely that a scheduling hierarchy 16 levels deep would ever be needed—the complex scenarios in Chapter 7 required at most five levels.

10.3.3 Cache Effects and Context Switching

This section presents data supporting the view that the true cost of a context switch can be much larger than the cost of executing the context switch code path in the kernel, because a newly-running thread may be forced to re-establish its working set in the processor cache. This argument is relevant to the feasibility of HLS because increased context switch costs can hide overhead caused by hierarchical scheduling—a subject we will return to at the end of this section.

Figure 10.4 shows the results of an experiment designed to measure this effect. The independent variables in this experiment are the sizes of threads' working sets and the scheduler that the threads belong to. Schedulers used in this experiment were the released version of the native Windows 2000 scheduler and the HLS proportional share scheduler—these were the schedulers with the shortest and longest median context switch times in Table 10.2.

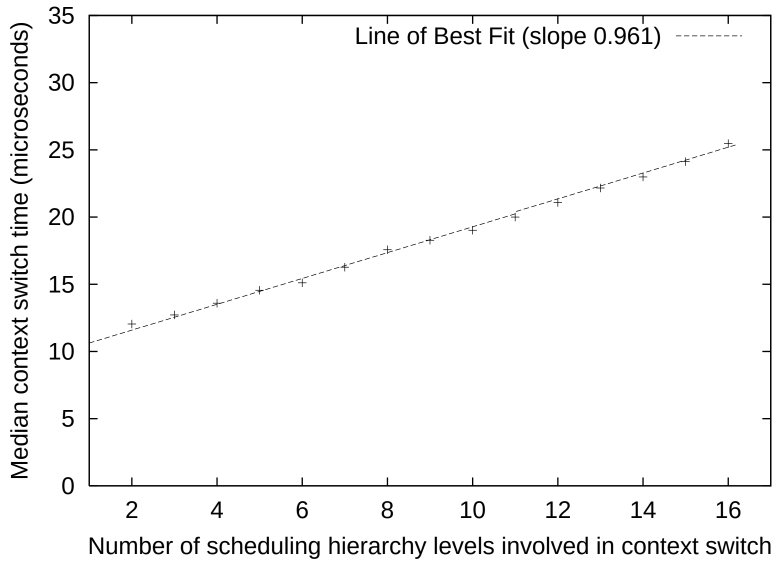

Figure 10.3: Thread context switch time as a function of hierarchy depth

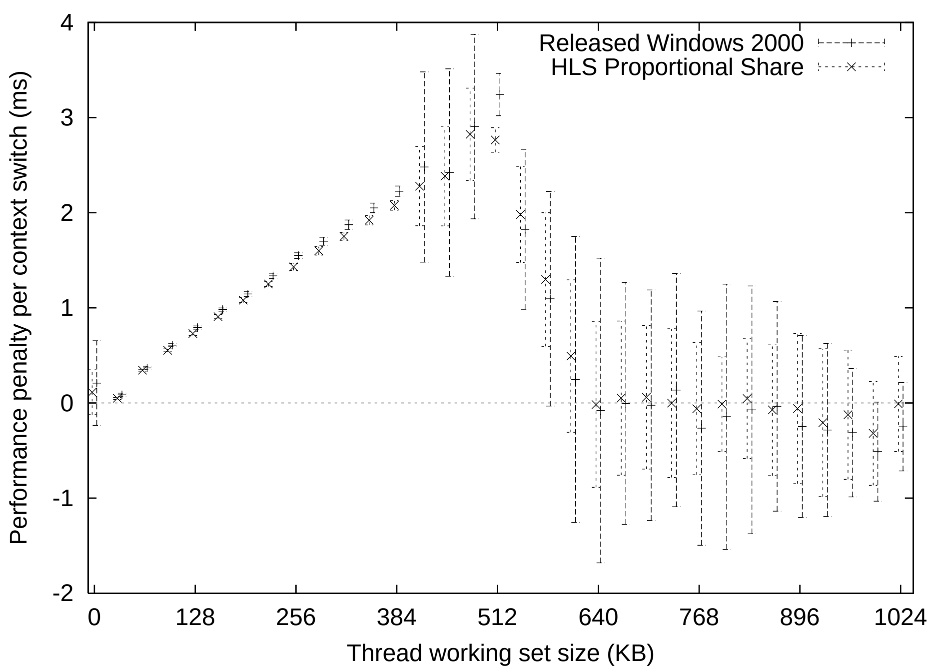

Figure 10.4: Thread performance penalty per context switch as a function of working set size

The work performed by each thread was to repeatedly cycle through a working-set-sized array by reading the value of an array location, performing a simple operation on it, and then writing the result back into the array. Since the level two (L2) cache on the Pentium III processor used for this experiment is 512 KB, a thread with working set size of 512 KB could be said to be making maximally effective use of the cache. The data points in Figure 10.4 represent working sets of sizes 0 KB through 1024 KB, in 32 KB increments. The data points for the two schedulers are offset slightly in the figure only to allow their error bars to be seen clearly: adjacent points represent identical working set sizes.

To generate each data point and the associated error bar, the amount of work performed by a single thread with a given working set size was compared with the aggregate amount of work performed by ten threads, each having a working set of the same size. The time-sharing schedulers caused context switches when the ten threads were running but not when one thread was running. The difference between the work performed by one and ten threads was assumed to be caused by context switch overhead. Let T be the amount of time the threads ran for, W_1 be the amount of work performed by one thread during the experiment, W_{10} be the total amount of work performed by ten threads, and N_{10} be the number of context switches between the ten threads. Then, the following equation describes the performance penalty C per context switch:

$$C = \frac{T(W_1 - W_{10})}{W_1 N_{10}} \tag{10.1}$$

The experiment was repeated 20 times, and the t-test [32, pp. 208–211] was used to determine significance of the results. The t-test is used to determine if the difference between two alternatives is significant. In this case it was used to decide if the aggregate throughput of ten threads was significantly less than that of one thread. For working set sizes of 32 KB through 544 KB, significant differences were found. We do not have a good explanation for why the data became so noisy for working set sizes above 544 KB, but there are several factors that could have caused variation between runs, such as *translation lookaside buffer* (TLB) effects and page coloring effects.

Another artifact of Figure 10.4 that we do not have a good explanation for is the fact that in the 32 KB–384 KB range, the performance penalty per context switch is slightly larger for the released Windows 2000 scheduler than it is for the HLS proportional share scheduler. This is almost certainly not due to a difference in the schedulers, but may again be caused by page coloring and TLB effects.

For working sets between 32KB and 384KB, the increase in thread performance penalty is very close to being linear in the size of threads' working sets. A line of best fit through the points in this region of the graph indicates that each additional KB of working set size increases the context switch performance penalty of a thread by 5.57 μs. If it takes the processor 5.57 μs to read 1 KB of data, 179 MB can be read in one second. This number is in the same order of magnitude as the memory bandwidth that the Stream Benchmark [57] reports on the test machine: 315 MB/s.

If we take the additional cost of a HLS context switch to be the difference between the median context switch time for the rebuilt Windows 2000 kernel and the median context switch time for the HLS time-sharing scheduler, 4.35 μs, then this cost is exceeded by the context switch performance penalty by an order of magnitude when each thread has a working set size of 8 KB or more, and it is exceeded by two orders of magnitude when threads have working sets of 79 KB or more. Furthermore, for threads whose working set is the same size as the L2 cache, the performance penalty

Clock interrupt frequency (Hz)	% Overhead (warm)	% Overhead (cold)
256	-0.102 ±0.20	0.0989±0.058
512	-0.141 ±0.18	0.260 ±0.040
1024	-0.00256±0.18	0.520 ±0.057
2048	0.387 ±0.18	1.12 ±0.054
4096	1.09 ±0.18	2.28 ±0.057
8192	2.74 ±0.17	3.69 ±0.040

Table 10.3: Reduction in application throughput due to clock interrupts as a function of frequency and cache state

of re-establishing the working set can be almost 3 ms—an enormous penalty when compared to the in-kernel context switch cost on the order of 10 μs.

In 1991 Mogul and Borg [62] showed that the cache performance cost of a context switch could dominate overall context switch performance. Furthermore, they speculated that in the future the increasing cost of memory accesses in terms of CPU cycle times would make the impact of the cache performance part of context switch cost increasingly dominant. They appear to have been correct.

10.4 Overhead of High-Frequency Timer Interrupts

The HLS prototype achieves fine-grained scheduling by increasing the frequency of the periodic clock interrupt in Windows 2000. The default clock interrupt frequency of the hardware abstraction layer (HAL) used by HLS is 64 Hz, with frequencies up to 1024 Hz supported. At 64 Hz the minimum enforceable scheduling granularity is 15.6 ms, and at 1024 Hz it is just under 1 ms. We modified HALMPS—the default HAL for multiprocessor PCs—to add support for increasing the clock frequency to 8192 Hz, achieving a minimum enforceable scheduling granularity of about 122 μs.

Table 10.3 shows the overhead caused by clock interrupts as a function of frequency. These numbers were taken by measuring the throughput of a test application that was allowed to run for ten seconds at each frequency. Twenty repetitions of the experiment were performed. The "warm cache" workload touched very little data, and therefore when the clock interrupt handler ran, its code and data were likely to be in the L2 cache of the Pentium III. The "cold cache" workload consisted of performing operations on an array 512 KB long (the same size as the L2 cache) in an attempt to flush the clock interrupt instructions and data out of the L2 cache.

The t-test was used to decide if application throughput was different between the baseline, with clock interrupts arriving at the default of 64 Hz, and the cases where the frequency was higher. If the difference plus or minus the confidence interval contains zero, the difference is not considered to be significant. The table shows that for the warm cache case, interrupt frequencies up to and including 1024 Hz do not cause statistically significant application slowdown, and that application slowdown at 8192 Hz is less than 3%. In the cold cache case, each frequency between 256 Hz and 8192 Hz caused a statistically significant slowdown, with a maximum slowdown of less than 4%.

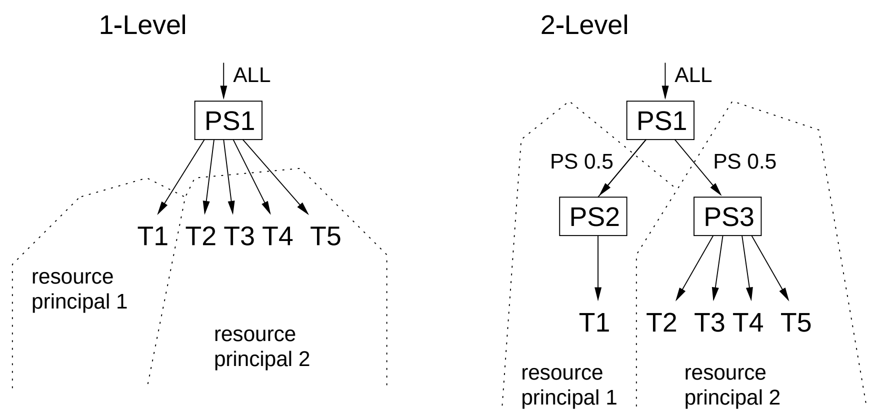

Figure 10.5: The 1-level hierarchy does not isolate resource principals, but the 2-level hierarchy does

This experiment shows that application slowdown due to high-frequency clock interrupts is not severe. Even so, a precisely settable interrupt facility would be a desirable feature in Windows 2000 since it would allow microsecond-accurate interrupts that do not cause any overhead when the interrupts are not required. Since HALMPS uses the *real-time clock*—a hardware device found on all modern PC motherboards—to generate clock interrupts, the *advanced programmable interrupt controller* (APIC)—another device found on all modern PCs—is left unused and a device driver could be written that uses the APIC to provide precise interrupts. This has been done for other general-purpose operating systems such as Linux [80].

10.5 Solving Problems Using Hierarchical Scheduling

Previous sections in this chapter argued that HLS is feasible. This section demonstrates the usefulness of HLS by presenting quantitative data about the performance of application threads being scheduled using HLS. In each of the two examples in this section, HLS schedulers are used to provide scheduling behaviors that cannot be provided using a traditional time-sharing scheduler. In other words, we show quantifiable benefits from sophisticated schedulers. Various schedulers that have been presented in the literature can provide one or the other of the scheduling behaviors described in these scenarios, but very few can provide both of them.

10.5.1 Isolation of Resource Principals

Hierarchical isolation of resource principals can be desirable when a machine is shared between users, administrative domains, or accounting domains. Figure 10.5 shows two scheduling hierarchies. The single-level scheduling hierarchy on the left is analogous to the schedulers in general-purpose operating systems: it considers each individual thread to be a resource principal, and allocates processor time to them accordingly. The scheduler on the right allocates a fair share of the processor to each of two resource principals.

Table 10.4 presents performance data gathered over 10-second runs using scheduling hierar-

	N_1	$\%_1$	N_2	$\%_2$
	1	50.0	1	50.0
without	1	19.9	4	80.1
isolation	1	6.10	16	93.9
(1-level)	1	1.56	64	98.4
	1	0.313	256	99.7
	1	50.0	1	50.0
with	1	50.0	4	50.0
isolation	1	50.1	16	49.9
(2-level)	1	50.1	64	49.9
	1	50.2	256	49.8

Table 10.4: Aggregate performance of isolated and non-isolated resource principals

chies like the ones depicted in Figure 10.5. The number of threads belonging to resource principal 1, N_1, is always equal to one in this experiment. The total fraction of the CPU allocated to the thread belonging to resource principal 1 is $\%_1$. Similarly, N_2 is the number of threads belonging to resource principal 2 and $\%_2$ is the total amount of CPU time allocated to them.

When both principals' threads are scheduled using a single proportional share scheduler, each thread receives a roughly equal share of the CPU. This means that principal 1's thread can be almost completely starved, receiving only 1.56% of the CPU when principal 2 creates 64 CPU-bound threads. Obviously, the act of creating a large number of CPU-bound threads does not have to be performed deliberately by a human operating as principal 2: they could be created by a malicious application or as the result of a programming error.

The bottom part of Table 10.4 confirms that when threads belonging to principals 1 and 2 are assigned to separate schedulers, each of which is given a proportional share guarantee, principal 2 cannot cause a thread belonging to principal 1 to be starved. For example, even when there are 256 CPU-bound threads that belong to principal 2, principal 1 receives approximately 50% of the CPU. This is the desired behavior. There is an apparent trend towards the bottom of Table 10.4 for principal 2 to receive slightly less than half of the CPU time; for example, 49.8% of the total for the 256-thread case. This is caused by overhead in principal 2's proportional share scheduler, which, as we mentioned in Section 10.3, is not implemented efficiently. Also, principal 1 is not receiving more than 50% of the total amount of CPU time, but rather, is performing more than 50% of the total amount of work done by both principals.

10.5.2 Scheduling a CPU-Bound Real-Time Application

Time-sharing schedulers on general-purpose operating systems can effectively schedule a single real-time application along with interactive and background tasks by running the real-time task at the highest priority. However, this simple method fails when the real-time application contains CPU-bound threads. The frame rendering loops in virtual environments and simulation-based games are CPU-bound because they are designed to adaptively provide as many frames per second

	app. guarantee	$\%_a$	FPS	misses	$\%_b$
0 background threads	RESBH 10 33	32.6	32.6	0	—
	RESBS 10 33	100.0	100.0	0	—
	NULL	100.0	100.0	0	—
1 background thread	RESBH 10 33	32.6	32.6	0	67.3
	RESBS 10 33	67.3	67.3	0	32.6
	NULL (high pri.)	96.7	96.7	6	3.26
	NULL (default pri.)	49.9	49.9	290	50.0
	NULL (low pri.)	3.11	3.11	985	96.9
10 background threads	RESBH 10 33	32.5	32.5	0	67.3
	RESBS 10 33	38.8	38.8	0	61.1
	NULL (high pri.)	68.9	68.9	10	31.0
	NULL (default pri.)	9.58	9.58	772	90.3
	NULL (low pri.)	3.57	3.57	888	96.4

Table 10.5: Performance of a CPU-bound real-time application with contention and various guarantees

(FPS) as possible; this makes it effectively impossible for them to gracefully share the processor with background applications when scheduled by a traditional time-sharing scheduler.

In the experiment described in this section, a synthetic test application is used instead of an actual application because the test application is self-monitoring: it can detect and record gaps in its execution, allowing the success or failure of a particular run to be ascertained. The situation described here is real, however, and anecdotal experience suggests that to successfully run a game under Windows 2000 using the default priority-based scheduler, it is necessary to avoid performing any other work on the machine while the game is being played, including background system tasks such as one that periodically indexes the contents of the file system.

The synthetic "virtual environment" application used in this experiment requires 10 ms of CPU time to render a single frame. The average frame rate must not fall below 30 FPS and furthermore, there must not be a gap of more than 33 ms between any two frames. If such a gap occurs, the test application registers the event as a deadline miss. The practical reason for this requirement is that in addition to degrading the visual experience, lag in virtual environments and games can lead to discomfort and motion sickness. Frame rates higher than 30 FPS are acceptable and desirable. Since each frame takes 10 ms to render, the hardware can render at most 100 FPS.

Table 10.5 shows the results of running the real-time application and background threads concurrently using different schedulers. Each line in the table describes the outcome of a single 30-second run. The percentage of the CPU received by the real-time application is $\%_a$ and the average number of frames per second produced by the real-time application is FPS (which is equal to the percentage because it takes 10 ms, or 1% of 1 s, to produce each frame). The number of times the real-time application failed to produce a frame on time during the experiment is listed under "misses" and finally, the total percentage of the CPU received by background threads, if any, is $\%_b$.

The top three lines in the table show the results of running the application with no background work. In this case a soft CPU reservation and no CPU reservation are equivalent: each allows the

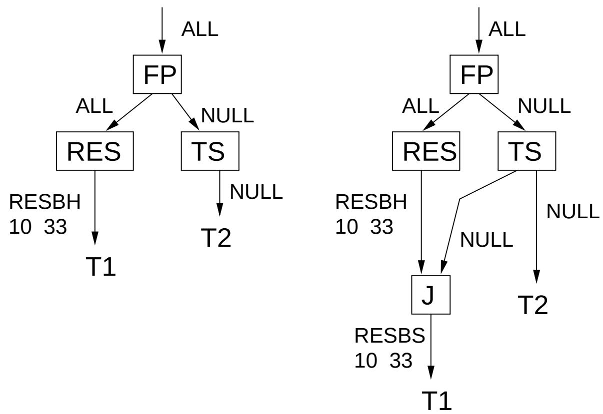

Figure 10.6: Scheduling hierarchies used to provide hard (left) and soft (right) CPU reservations for the experiment in Section 10.5.2

application to use 100% of the processor. A hard CPU reservation of 10 ms / 33 ms permits the application to achieve slightly more than its minimum frame rate (the application receives slightly more than the amount of CPU time that it reserved because the enforceable scheduling granularity is limited). The scheduling hierarchies used to provide hard and soft reservations in this experiment are depicted in Figure 10.6.

The middle cluster of lines in Table 10.5 shows the effects of running the real-time application alongside a single CPU-bound background thread. In this experiment, background threads were always run at the default time-sharing priority. The scheduling behavior of the real-time application when given a hard CPU reservation is the same as in the case of no contention: it achieves slightly more than the minimum required frame rate, and misses no deadlines. Given a soft CPU reservation, the real-time application and the background thread share unreserved CPU time—this is the desired behavior. Without a CPU reservation, the real-time application is unable to gracefully share the machine with the background thread. When the real-time application is scheduled at a higher priority than the background thread, the background thread is not able to make much progress: it runs only occasionally, when its priority is boosted by the heuristic that we mentioned in Section 9.2.2, that prevents threads from being completely starved regardless of their priority. Notice that although the background thread only received about 3% of the CPU time during this test, it sill caused the real-time application to miss six deadlines. When the real-time application is run at the same priority as the background thread it is unable to meet its deadlines even though it is receiving more than enough processor time: the time-sharing scheduler time-slices between the two threads at a granularity too coarse to meet the scheduling needs of the real-time application. When the real-time application is run at a priority lower than that of the background thread, it is scheduled only occasionally and provides few frames.

The situation where there are ten background threads is similar to the situation where there is one: the real-time thread only meets all deadlines when given a CPU reservation.

In conclusion, only a soft CPU reservation provides the desired scheduling behavior in this example: a hard CPU reservation precludes opportunistic use of spare CPU time to provide a high frame rate, and no CPU reservation precludes sharing the machine with background threads.

10.6 Optimizing HLS

HLS was designed to use low-cost, constant-time operations when making scheduling decisions. Beyond this design, little effort was put into optimizing the HSI or the loadable schedulers. They could be optimized by:

- Removing the native Windows 2000 scheduler code and the data structures that it manipulates.
- Making the mapping from scheduler names to scheduler instances more efficient. This is currently performed with a linear lookup, which is potentially costly since there is a scheduler instance for each thread in the system. This lookup would be very fast if it were implemented with a hash function or a binary search. The overhead of this lookup is incurred only when a named scheduler instance must be found; for example, to begin a CPU reservation. Lookups are not part of normal scheduling decisions.
- Identifying a “core” scheduling hierarchy that will always be present on a particular system, and then statically optimizing the core hierarchy using a component optimizer such as the flattening tool developed for the Knit component composition system [74]. Also, some of the techniques that have been developed to optimize layered communication protocols could be applied to hierarchical schedulers. For example, a tool was developed to remove virtual function calls in the Click modular router [44] and an optimization approach based on a theorem prover was used to optimize a graph of communicating components in Ensemble [54].
- Using dynamic compilation techniques to eliminate virtual function calls and otherwise regain optimization opportunities lost due to separate compilation of schedulers.

In principle, there does not appear to be any reason that context switches involving a single hierarchical scheduler should be any more expensive than context switches in the released Windows 2000 scheduler. Scheduling decisions involving many loadable schedulers will unavoidably add some overhead.

10.7 Conclusion

This dissertation supports the thesis that HLS is feasible and useful, and this chapter contained material supporting both points. Data showing that HLS mode-change operations are cheap and that HLS causes a modest increase in context switch time support the feasibility argument, and quantitative data about application performance when scheduled by loadable schedulers supports the argument for the usefulness of HLS.

Chapter 11

Augmented CPU Reservations

The problem that this chapter addresses is that low-level system activity in a general-purpose operating system can adversely affect the predictable scheduling of real-time applications. In effect, the OS "steals" time from the application that is currently scheduled. Two schedulers that provide *augmented CPU reservations* are described and evaluated; they increase the predictability of real-time applications in the presence of *stolen time*

To strengthen the case for augmented CPU reservations, we have performed a study of the amount of CPU time that can be stolen by a number of different device drivers in real-time versions of Linux and Windows 2000. We learned, for example, that the default configuration of the IDE disk driver in Linux can steal close to 50% of a CPU reservation.

Augmented reservations are just one way of coping with stolen time. Section 11.9 presents a comparison and evaluation of different methods of coping with stolen time.

11.1 Augmented Reservations and Guarantees

The work on guarantees that was described in Chapter 5 assumed that stolen time due to operating system overhead does not interfere with guarantees. This assumption is justified because most operating system events such as scheduling decisions and interrupts take on the order of microseconds or tens of microseconds, while multimedia and other soft real-time applications generally require processing on a granularity of milliseconds or tens of milliseconds. Furthermore, due to caching, branch prediction, pipelining, and other effects, it can be very difficult to establish a tight bound on the worst-case execution time for a piece of code. It is assumed that applications will have to overestimate their CPU requirements slightly, and that stolen time can be "hidden" in the slack time.

In certain cases—the ones that we focus on in this chapter—stolen time significantly interferes with applications and it must be taken into account if deadlines are to be met. Although it may be possible to revise the definitions of guarantees to be probabilistic in order to reflect the possibility of stolen time, this would complicate analysis of scheduler composition (even in the common case where there is little stolen time), and it is not clear that stolen time can be bounded or that it follows any standard probability distribution.

The approach taken in this chapter is to leave the integration of stolen time into the guarantee framework as future work. Augmented CPU reservations, then, are not a kind of guarantee that is

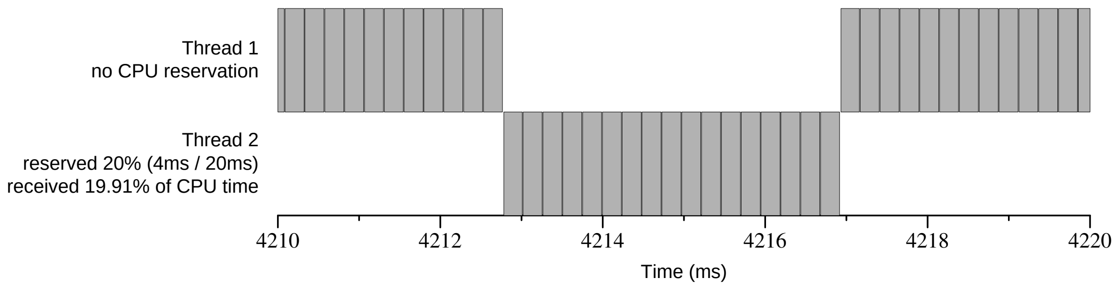

Figure 11.1: Execution trace of a CPU reservation functioning correctly on an otherwise idle machine

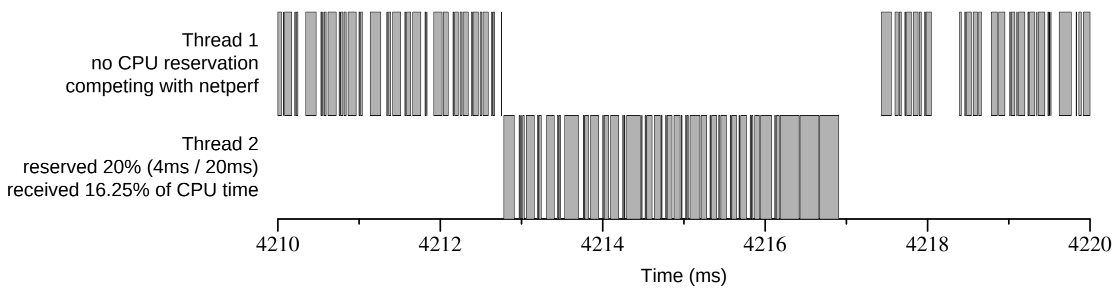

Figure 11.2: Gaps in Thread 2's CPU time indicate time being stolen from a CPU reservation by the kernel

different from the reservation guarantees presented in Chapter 5. Rather, they are the same type of guarantee, but they allow an assumption made by the reservation scheduler, that there is a negligibly small amount of stolen time, to be relaxed.

11.2 Motivation: Time Stolen by the Network Stack

Figures 11.1 and 11.2 illustrate the effects of stolen time. Each figure shows an actual execution trace of a CPU reservation that was granted by Rez. In both figures Thread 2 has been granted a CPU reservation of 4 ms / 20 ms. In the center of each figure is a single 4 ms block of time that Rez allocated to Thread 2. Each millisecond appears to be divided into roughly four chunks because the clock interrupt handler periodically steals time from the running application—we ran the clock at 4096 Hz in order to approximate precisely scheduled interrupts, which are not available in Windows 2000.

In Figure 11.1 our test application has the machine to itself, and Thread 1 uses up all CPU time not reserved by Thread 2. In Figure 11.2, the test application is running concurrently with an instance of Netperf, a network bandwidth measurement application that is receiving data from another machine over 100 Mbps Ethernet. Thread 1, which is running at low priority, gets less CPU time than it did in Figure 11.1 because it is sharing the processor with Netperf. However, Netperf does not get to run during the block of time reserved for Thread 2. There are gaps in the CPU time received by Thread 2 because the machine continues to receive data even when the Netperf application is not being scheduled, and the kernel steals time from Thread 2 to process this data. This stolen time causes Thread 2 to receive only about 82% of the CPU time that it reserved. A real-time application running under these circumstances will have difficulty meeting its deadlines.

The root of the problem is that although Rez ensures that Thread 2 will be scheduled for a 4 ms interval (or for several intervals adding up to 4 ms) during each 20 ms period, it is not necessarily the case that Thread 2 will get to execute for the entire 4 ms—stolen time is invisible to the thread scheduler. To address this problem we have designed and implemented two novel schedulers, Rez-C and Rez-FB, that provide *augmented CPU reservations*—they actively measure stolen time and counteract its effects, allowing deadlines to be met even when the OS steals a significant amount of CPU time from a real-time application. Rez-C gives applications the opportunity to "catch up" after time has been stolen from them, and Rez-FB uses a feedback loop to ensure that in the steady state applications actually receive the amount of CPU time that they have reserved.

11.3 Characterization of Stolen Time

We define *stolen time* to be CPU time spent doing something other than running the currently scheduled application or performing services on its behalf. Time is stolen because *bottom-half device driver processing* in general-purpose operating systems is assigned a statically higher priority than any application processing, and because this time is not accounted for properly: it is "charged" to the application that happens to be running when a device needs service.

Bottom-half processing occurs in the context of interrupts and *deferred procedure calls*. Interrupts are hardware-supported high-priority routines invoked when an external device requests service. Deferred procedure calls (DPCs), which are analogous to *bottom-half handlers* in Linux and other Unix variants, were designed to give device drivers and other parts of the kernel access to high-priority, lightweight asynchronous processing outside of interrupt context [77, pp. 107–111]. DPCs were identified as potential problems for real-time tasks on Windows NT by Ramamritham et al. [72].

In effect, Windows 2000 and similarly structured operating systems contain not one but three schedulers. Interrupts take precedence over all other processing and are scheduled by a fixed-priority preemptive scheduler that is implemented in hardware. DPCs take precedence over all thread processing and are scheduled by a simple FIFO scheduler. Finally, applications are scheduled at the lowest priority by the OS thread scheduler (which is normally thought of as "the scheduler").

11.4 Approach

Rez, the reservation scheduler that we described in Section 9.3.2, is the basis for the augmented reservation schedulers presented in this chapter. Rez suffers from reduced predictability because bottom-half mechanisms in Windows 2000 can steal time from real-time applications. Our approach to improving predictability is called *augmented reservations*. The basis of this approach is to give the reservation scheduler access to fine-grained accounting information indicating how long the kernel spends running DPCs, allowing it to react accordingly. To this end we implemented two additional versions of the Rez scheduler called Rez-C and Rez-FB; they are described in Sections 11.5.2 and 11.5.3, respectively.

The scheduling hierarchy that we used for all Windows 2000 experiments reported here runs one of the variants of Rez at the highest priority, and runs a priority-based timesharing scheduler when Rez has no threads to run. Rez has an admission controller that rejects any requests for reservations that would cause the cumulative CPU requirements over all reservations to exceed a fixed fraction of the capacity of the CPU. To prevent reservations from starving the timesharing scheduler this parameter can be set to a value less than 1.0. We currently use 0.85; the value chosen should reflect the relative importances of the time-sharing and real-time classes of applications on a particular system.

11.5 Coping with Stolen Time

Once a real-time scheduling abstraction such as CPU reservations has been implemented within a general-purpose operating system, the ways to increase predictability with respect to stolen time form a continuum:

1. To avoid further modifications to the core OS, but to manually move device driver code out of time-stealing bottom-half contexts.

2. To instrument stolen time and feed the resulting information into the real-time scheduler to allow it to compensate.

3. To modify bottom-half mechanisms to put them under control of the scheduler, eliminating this source of stolen time.

The first option has been recently explored by Jones and Saroiu [41] in the context of a software modem driver. Jeffay et al. [34] chose the third option: they modified FreeBSD to perform proportional-share scheduling of network protocol processing. Our work is based on the second option. There are advantages and disadvantages to each approach, and we believe that there are situations in which each of them is appropriate. We will compare the approaches in detail in Section 11.9.

11.5.1 Measuring Stolen Time

Of the two sources of stolen time in Windows 2000, interrupts and DPCs, we have instrumented only DPCs. Although it would be straightforward to instrument hardware interrupt handlers as well, the time spent in DPCs serves as a useful approximation of the true amount of stolen time because interrupt handlers in Windows 2000 were designed to run very quickly: they perform critical processing and then enqueue a DPC to perform any additional work.

To instrument DPCs we added code to the beginning and end of the dispatch interrupt handler (a software interrupt handler that dequeues and runs DPCs) to query the Pentium timestamp counter (using the `rdtsc` instruction) which returns the number of cycles since the machine was turned on. By taking the difference between these values the system can accurately keep track of the amount of time spent running DPCs.

The interface to stolen time accounting is the `GetStolen()` function, which is available to code running in the Windows 2000 kernel; it returns the amount of stolen time since it was last called. In other words, it atomically reads and zeros the stolen time counter.

11.5.2 Rez-C: Increasing Predictability by Catching Up

Rez-C gives threads the opportunity to catch up when they have had time stolen from them. It does this by deducting from budgets only the actual amount of CPU time that threads have received, rather than deducting the length of the time interval that they were nominally scheduled for, which may include stolen time. For example, if a thread with a reservation of 4 ms / 20 ms is runnable and will have the earliest deadline during the next 4 ms, Rez-C schedules the thread and arranges to regain control of the CPU in 4 ms using a timer. When the timer expires, Rez-C checks how much time was stolen during the 4 ms interval using the `GetStolen()` function. If 1.2 ms were stolen, then Rez-C deducts 2.8 ms from the thread's budget. If the thread still has the earliest deadline, Rez-C arranges to wake itself up in 1.2 ms and allows the thread to keep running.[1]

Rez-C uses accounting information about stolen time at a low level, bypassing the admission controller. When there is not sufficient slack in the schedule, allowing a thread to catch up may cause other threads with reservations to miss their deadlines or applications in the timesharing class to be starved. These deficiencies motivated us to design Rez-FB.

11.5.3 Rez-FB: Increasing Predictability using Feedback Control

Our second strategy for coping with stolen time assumes that the amount of stolen time in the near future will be similar to what it was in the recent past. It uses a feedback loop to minimize the difference between the amount of CPU time that each application attempted to reserve and the amount of CPU time that it actually receives. There is an instance of the feedback controller for each thread that has a CPU reservation. The parameters and variables used by the feedback controller are:

- A set point R, the amount of CPU time that an application requested.
- A control variable C, the amount of CPU time reserved by Rez-FB on behalf of an application.
- A process variable P, the amount of CPU time that an application actually receives; this is calculated by summing the lengths of the time intervals during which the application was scheduled and subtracting the amount of time stolen during those intervals.
- A constant gain $G \leq 1$.

The feedback equation, which is evaluated for each reservation each time its period starts, is:

$$C_t = C_{t-1} + G(R - P_{t-1})$$

In other words, at the beginning of a reservation's period the amount of CPU time requested by Rez-FB on behalf of the application is adjusted to compensate for the difference between the desired and actual amounts of CPU time during the previous period. The gain helps prevent overshooting and can be used to change how aggressively the controller reacts to changing amounts of stolen time.

[1] This process does not lead to infinite regress because timer interrupts are quantized: Rez-C's minimum enforceable scheduling granularity is 242 μs. If Rez-C had access to precisely scheduled timer interrupts, it would still be forced to quantize timers to balance the overhead caused by excessive numbers of timer interrupts with the requirement for accurate scheduling.

Because Rez-FB applies accounting information to reservation amounts rather than budgets, it does not bypass the admission controller. Therefore, Rez-FB will not allow threads with CPU reservations to interfere with each other, or with time-sharing applications. On the other hand, Rez-FB reacts more slowly to stolen time than Rez-C, potentially causing applications to miss more deadlines when the amount of stolen time varies on a short time scale. The feedback controller currently used by Rez-FB is a special case of a PID (Proportional-Integral-Derivative) controller that contains only the integral term. In the future we may attempt to improve Rez-FB's response to changes in the amount of stolen time using more sophisticated controllers that have proportional and derivative terms as well.

11.5.4 Evaluating Rez-C and Rez-FB

In the next two sections we will evaluate and compare Rez-C and Rez-FB according to the following criteria:

- How well do augmented reservations help applications meet their deadlines under adverse conditions?
- How efficient are they in terms of run-time overhead?
- Do they display graceful behavior during overload?
- What are their costs in terms of application and system developer effort?

11.6 Experimental Evaluation of Rez-C and Rez-FB

11.6.1 Application Testing

The application scenarios that we will consider include the following elements: a general-purpose operating system, Windows 2000, that has been extended with Rez (as a control), Rez-C, or Rez-FB; a soft real-time application that uses a reservation to meet its periodic CPU requirements; and, a background workload that causes the OS to steal time from the real-time application.

11.6.2 Test Application

The soft real-time application used in our experiments is a synthetic test application. The important qualities of this application are: the ability to create multiple threads at different priorities, with optional CPU reservations; the ability to detect stolen time by measuring exactly when these threads are scheduled; and, for threads with reservations, the ability to determine if and when deadlines are missed.

To this end we started with a test application that was originally developed for the Rialto operating system at Microsoft Research and later ported to Rialto/NT [37]. For the work reported here we ported it to TimeSys Linux/RT and Windows 2000 + Rez, and gave it the capacity to detect deadline misses for threads with ongoing CPU reservations.

Rather than using the CPU accounting provided by the operating system[2] our test application repeatedly polls the Pentium timestamp counter. When the difference between two successive reads is longer than 2.2 μs, time is assumed to have been stolen from the application between the two reads. This number was experimentally determined to be significantly longer than the usual time between successive reads of the timestamp counter and significantly shorter than common stolen time intervals. The duration of the stolen time interval is taken to be the difference between the two timer values minus the average time spent on the code path between timer reads (550 ns). We believe that this provides a highly accurate measure of the actual CPU time received by the application.

Threads with reservations repeatedly check if the amount of wall clock time that has elapsed since the reservation was granted has passed a multiple of the reservation period. Each time this happens (in other words, each time a period ends) they register a deadline hit if at least the reserved amount of CPU time has been received during that period, or a deadline miss otherwise.

11.6.3 Test Workload

The workload used is an incoming TCP stream over 100 Mbps Ethernet (we characterize other sources of stolen time in Section 11.8). We chose this workload because it is entirely plausible that a desktop computer may be in the middle of a real-time task such as playing a game or performing voice recognition when high-bandwidth data (such as a file transfer) arrives over a home or office network.

To actually transfer data we used the default mode of Netperf [29] version 2.1, which establishes a TCP connection between two machines and transfers data over it as rapidly as possible.

11.6.4 Test Platform

Our test machine was a dual 500 MHz Pentium III with 256 MB of RAM. It ran in uniprocessor mode for all experiments. It was connected to the network using an Intel EtherExpress Pro/100B PCI Ethernet adapter. For all experiments in this section the machine ran one of our modified Windows 2000 kernels, and had a timer interrupt period (and therefore a minimum enforceable scheduling granularity) of 244 μs.

11.6.5 Statistical Methods

Unless otherwise stated, all data points in this section and in Section 11.8 are averages of ten 10-second runs. Confidence intervals were calculated at 95%.

11.6.6 Reducing Deadline Misses using Rez-C and Rez-FB

Figure 11.3 shows the number of deadline misses detected by our test application with a reservation of 4 ms / 20 ms under four different conditions:

1. Scheduled by Rez, on an otherwise idle machine.

[2] The statistical accounting performed by Windows 2000 and Linux is particularly inappropriate for monitoring the usage of threads that have CPU reservations: since accounting is periodic, the periods of reservations and accounting can resonate, causing threads to be drastically over- or under-charged for their CPU usage.

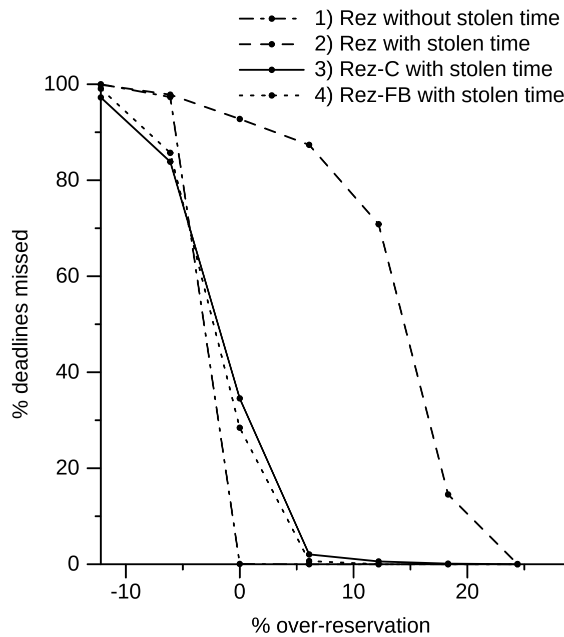

Figure 11.3: Predictability of Rez, Rez-C, and Rez-FB

2. Scheduled by Rez, while the test machine received a TCP stream over 100 Mbps Ethernet.
3. Scheduled by Rez-C, while the test machine received a TCP stream over 100 Mbps Ethernet.
4. Scheduled by Rez-FB, while the test machine received a TCP stream over 100 Mbps Ethernet.

To meet each deadline, the test application needed to receive 4 ms of CPU time during a 20 ms period. In order to demonstrate the effect of statically over-reserving as a hedge against stolen time, for each of the four conditions we had Rez actually reserve more or less than the 4 ms that was requested. So, if Rez were set to over-reserve by 50%, it would actually reserve 6 ms / 20 ms when requested to reserve 4 ms / 20 ms.

Line 1 shows that on an idle machine, any amount of under-reservation will cause most deadlines to be missed, and that no deadlines are missed by the test application when it reserves at least the required amount of CPU time. This control shows that Rez is implementing CPU reservations correctly.

Line 2 (the rightmost line on the graph) illustrates the effect of time stolen by network receive processing. To avoid missing deadlines, Rez must over-reserve by about 24%. This is quite a large amount, and would not prevent the application from missing deadlines in the case that several drivers steal time simultaneously. In Section 11.9.1 we will argue that pessimistic over-reservation is not a good general solution to the problem of deadline misses due to stolen time.

Lines 3 and 4 are very similar, and show that both Rez-C and Rez-FB increase the predictability of CPU reservations when the OS is stealing time from applications. Notice that a small amount of over-reservation (about 6%) is required before the percentage of missed deadlines goes to nearly

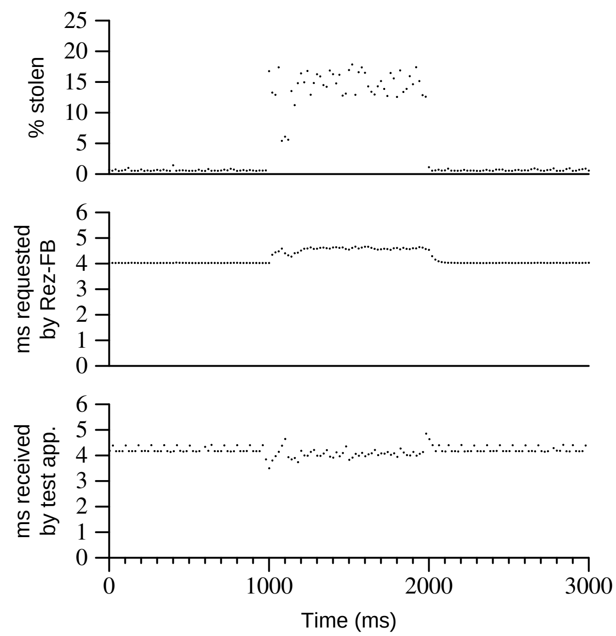

Figure 11.4: Performance of Rez-FB under changing load: the test machine received a TCP stream over 100 Mbps Ethernet between times 1000 and 2000. Each data point represents a single 20 ms period.

zero. Some deadlines are missed in the 0% over-reservation case because we only instrumented DPCs, and not hardware interrupt handlers.

We calculated confidence intervals for the data points in Figure 11.3 but omitted them from the graph because they were visually distracting. The 95% confidence interval was always within 3.4% of the means reported here, and was significantly less than that for most of the data points.

11.6.7 Response of Rez-FB to Variations in Load

Rez-C is very simple and retains no information about stolen time between periods of a reservation. However, as we described in Section 11.5.3, Rez-FB uses a feedback loop that compares its current performance to performance in the previous period. This raises questions about stability, overshoot, and reaction time. The intuition behind its feedback loop is straightforward and we believe that Rez-FB is stable and predictable, and that it will quickly converge on the correct amount of CPU time to allocate. However, we have performed an experiment under changing load in order to test this hypothesis. Figure 11.4 shows the response of Rez-FB to a 1-second burst of network traffic, which happens between times 1000 and 2000. The test application has created a single thread with a CPU reservation of 4 ms / 20 ms. So, $R = 4.0$ ms. This graph contains no confidence intervals since the data come from a single execution. Each point on the graph represents a single 20 ms period.

The top graph in Figure 11.4 shows the amount of stolen time reported to Rez-FB by each call to `GetStolen()`, expressed as a percentage of 4 ms. It shows that there is a small base amount of time stolen by background system activity, and that stolen time increases dramatically during network activity. This is exactly what we would expect. The middle graph shows C_t, the amount of CPU time requested by Rez-FB during each period. As expected, Rez-FB requests more CPU time when more time is being stolen by the kernel. Although the amount of stolen time at the start of Netperf's run is noisy (probably due to TCP slow start and other factors), its finish is abrupt and around time 2000 C_t drops from its elevated level back to about 4.0, cushioned by the gain G. Although we tried several different values for G, all experiments reported here used a value of 0.5. We did not look into the sensitivity of this parameter in detail, but values between 0.5 and 1 appeared to produce about the same number of missed deadlines.

The bottom graph shows P_t, the actual amount of CPU time received by our test application during each period. While the machine is quiet (in ranges 0-1000 and 2000-3000) the values of P_t are quantized because the scheduling enforcement granularity is limited to 244 μs. A deadline is missed whenever P_t is below 4.0; this happened 22 times during the test. This is due to unaccounted stolen time from the network interrupt handler and also to lag in the feedback loop. While missing 22 deadlines may be a problem for some applications, this is significantly better than missing most of the 500 deadlines between times 1000 and 2000, as would have happened with Rez under the same conditions. To support applications that cannot tolerate a few missed deadlines, the system would need to instrument stolen time more comprehensively or statically over-reserve by a small amount.

11.6.8 Run-Time Overhead of Rez-C and Rez-FB

Both Rez-C and Rez-FB add very little overhead to the scheduler. The overhead of instrumenting DPCs (incurred each time the kernel drains the DPC queue) is twice the time taken to read the Pentium timestamp counter and write its result to memory, plus the time taken by a few arithmetic operations. Similarly, the overhead of the `GetStolen()` call is the time taken to run a handful of instructions.

To verify that the augmented reservation schedulers add little overhead we measured how much CPU time was lost to a reservation running under Rez-C and Rez-FB as compared to the basic Windows 2000 + Rez. This revealed that Rez-C adds 0.012%$\pm$0.0028 overhead and Rez-FB adds 0.017%$\pm$0.0024 overhead. We would have been tempted to assume that these differences were noise but the confidence intervals indicate that they are robust.

11.7 Other Criteria for Evaluating Rez-C and Rez-FB

11.7.1 Behavior During Overload

As we mentioned in Section 11.5.2, Rez-C has two main problems. First, its catchup mechanism can cause threads with CPU reservations to miss their deadlines, and second, when the total amount of reserved time plus the total amount of stolen time exceeds the capacity of the processor, timesharing threads will be starved. The first problem is inherent in the design of Rez-C, but the second could be helped by giving a CPU reservation to the entire class of time-sharing applications (this is easy in a system that supports hierarchical schedulers). While this would not prevent overload, giving the timesharing scheduler a reservation would ensure that it is not completely starved because there would be at least some times at which it had the earliest deadline.

In Rez, Rez-C, and Rez-FB it would be a good idea to give the timesharing scheduler a reservation even when there is no risk of overload, to ensure that timesharing applications are scheduled often enough that their response to user input appears to be smooth rather than jumpy. For example, a real-time application with a reservation of 0.5 s / 1 s would currently make the system difficult to use by allowing timesharing applications (such as the user interface) to run only every half-second. A CPU reservation (of 7 ms / 50 ms, for example) for the timesharing scheduler would keep non-real-time applications responsive to user input. In other words, humans introduce soft deadlines into the system; missing these deadlines will result in bored or irritated users.

Clearly, if timesharing applications were given a reservation we would want them to be able to use idle time in the schedule in addition to their reserved time. While this would not be difficult to implement, Rez currently does not allow a thread (or in this case, a collection of threads) with a CPU reservation to use more processor time than was reserved. Some reservation schedulers, such as the one in the portable resource kernel [68], have the ability to make CPU reservations either *hard*, guaranteeing both a minimum and maximum execution rate, or *soft*, guaranteeing only a minimum execution rate for applications that can make use of extra CPU time. This is a useful specialization, and implementing it in Rez, Rez-C, and Rez-FB is part of our future plans.

We have seen that Rez-C does not gracefully handle overload caused by stolen time. Rez-FB is better in this respect: it does not allow applications with CPU reservations to interfere with each other or with timesharing applications. However, overload in Rez-FB raises policy issues. Recall that overload in Rez-FB can be induced either by stolen time or by genuine application requests. Although it currently treats these cases identically (by rejecting all requests that would make system load exceed a fixed percentage), a more sophisticated admission control policy may be desirable. For example, assume that an important application is having time stolen, causing Rez-FB to increase the amount of its reservation. If this request would result in overload, perhaps instead of rejecting it Rez-FB should revoke some CPU time from a less important application, or one that can tolerate missed deadlines. Clearly there are many policy decisions that could be made here; we have not explored them.

11.7.2 Costs Imposed on the System and Application Developer

Neither Rez-C nor Rez-FB imposes additional cost on the developers of real-time applications. Developers should only notice reservations becoming more predictable while a machine is receiving asynchronous data.

Both of these schedulers were fairly easy to implement once the basic Rez scheduler was working. Because Rez-C and Rez-FB make use of existing abstractions (budgets and reservation amounts) they each required only about 40 lines of code to be added to the basic Rez scheduler. The code to instrument DPCs added about 50 lines of assembly code to Windows 2000's dispatch interrupt handler.

11.7.3 Rez, Rez-C, and Rez-FB Compared

We have shown that stolen time can cause tasks with CPU reservations granted by Rez to miss their deadlines. Rez-C and Rez-FB, on the other hand, allow applications to receive approximately the same scheduling behavior when time is being stolen from them as they do on an idle machine. Rez-C behaves poorly when subjected to overload caused by stolen time, but we do not yet have enough experience with it to know if this is a significant problem in practice. Of the three, Rez-FB appears to be the most promising: it works well and does not have the disadvantages of Rez-C.

11.8 Stolen Time in Other Systems and by Other Devices

In Figure 11.3 we showed that network activity can significantly increase the number of deadlines an application misses despite the CPU scheduler's best efforts. In this section we provide additional motivation for augmented CPU reservations by presenting the results of a study of the amount of time that can be stolen by the Linux and Windows 2000 kernels when they receive asynchronous data from a number of different external devices. For the Linux tests we used TimeSys Linux/RT [86], which adds resource kernel functionality [68] such as CPU reservations and precise timer interrupts to Linux.

Our goal was not to compare the real-time performance of Linux/RT and Windows 2000 + Rez, but rather to shed light on the phenomenon of stolen time and to find out how much time can be stolen by the drivers for various devices on two completely different operating systems. Indeed, these results will generalize to any operating system that processes asynchronous data in high-priority, bottom-half contexts without proper accounting. As far as we know, these are the first published results of this type.

For Linux tests we used TimeSys Linux/RT version 1.1A, which is based on version 2.2.14 of the Linux kernel. All Linux tests ran on the same machine that ran all of our Windows 2000 tests (a dual 500 MHz Pentium III booted in uniprocessor mode).

11.8.1 Effect of Reservation Amount on Time-Stealing by Network Processing

Figure 11.5 shows how the amount of time stolen from a CPU reservation by network receive processing changes with the amount of the reservation. The test application reserved between 1 ms and 10 ms out of 20 ms. The reason that the proportion of stolen time decreases as the size of the block of reserved time increases can be seen by looking closely at Figure 11.2: towards the end of the reserved block of time (after time 4216) there is little stolen time. This is because the Netperf application does not get to run during time reserved by the real-time application; therefore, kernel network buffers are not drained and packets are not acked, causing the sender to stop sending after a few milliseconds.

Although Figure 11.5 would appear to indicate that Linux processes incoming network data more efficiently than Windows 2000, no such inference should be drawn. The bandwidth received by our test machine while running Windows 2000 + Rez was around 93 Mbps and the bandwidth received while running Linux/RT was only about 81 Mbps. We do not have a good explanation for this, although we do know that the low network performance is not an artifact of Linux/RT—we observed the same low bandwidth while running the Linux kernel that shipped with Redhat 6.2. The sender and receiver were directly connected by a fast Ethernet switch and the machine sending the data was a 400 MHz Pentium II running Windows NT 4.0.

11.8.2 Hard Disk Controllers

Table 11.1 shows the amount of time that was stolen from CPU reservations of 4 ms / 20 ms by OS kernels as they processed data coming from hard disks. We show measurements for both Linux/RT

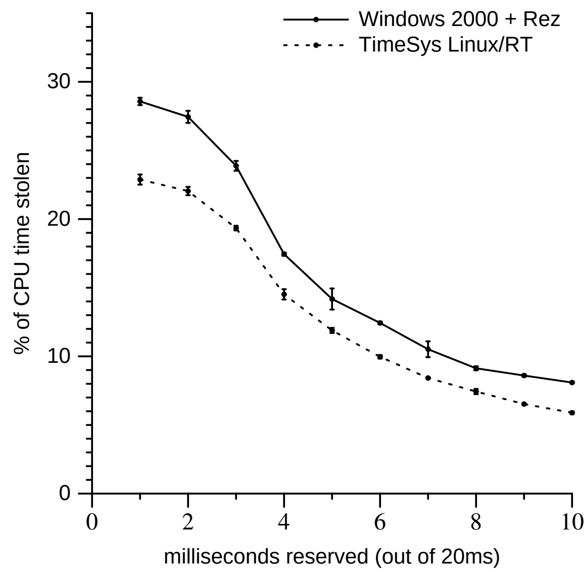

Figure 11.5: Time stolen by the kernel to process an incoming TCP stream. Reservations have 20 ms periods and varying amounts.

Windows 2000 + Rez	IDE w/ DMA	0.78%±0.052
	IDE w/ PIO	n/a
	SCSI w/ DMA	0.55%±0.026
	SCSI w/ PIO	n/a
Linux/RT	IDE w/ DMA	1.1%±0.28
	IDE w/ PIO	49%±3.5
	SCSI w/ DMA	0.74%±0.20
	SCSI w/ PIO	n/a

Table 11.1: Amount of time stolen from a CPU reservation by disk device drivers

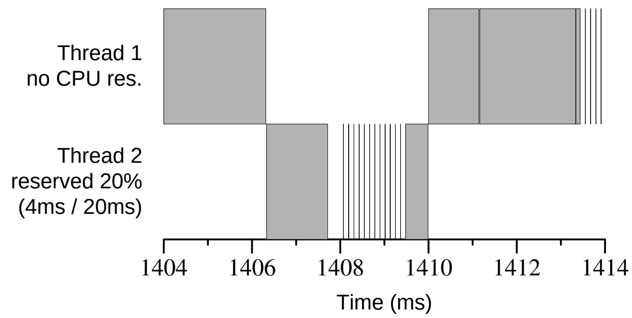

Figure 11.6: Execution trace showing time stolen from a CPU reservation in Linux/RT by the IDE disk driver operating in PIO mode

and Windows 2000 + Rez, for a SCSI and an IDE disk, and using both direct memory access (DMA) and programmed I/O (PIO) to move data to the host, when possible.

The SCSI disk used in this test is a Seagate Barracuda 36, connected to the host over an Ultra2 SCSI bus; it can sustain a bandwidth of 18.5 MB/s while reading large files. The IDE disk is an older model (a Seagate 1270SL) that can sustain a bandwidth of only about 2.35 MB/s.

From this table we conclude that disk transfers that use DMA cause the OS to steal only a small amount of time from real-time applications (the confidence intervals indicate that the differences, though small, are real). However, even a slow disk can severely hurt real-time performance if its driver uses PIO: in our test it caused the Linux kernel to steal nearly half of the CPU time that a real-time application reserved. Therefore, it is imperative that real-time systems avoid using PIO-based drivers for medium- and high-speed devices. The large gap in Thread 2's block of reserved time in Figure 11.6 illustrates the problem.

Unfortunately, Linux distributions continue to ship with PIO as the default data transfer mode for IDE disks. For example, we recently installed Redhat Linux 7.0 on a new machine equipped with a high-performance IDE disk. Due to overhead of PIO transfers, the machine was barely usable while large files were being read from the hard disk. Interactive performance improved dramatically when we turned on DMA for the hard disk using the `hdparm` command. Windows 2000 uses DMA by default for both SCSI and IDE disks.

11.8.3 Software-Based Modems

Software-based modems contain minimal hardware support: they perform all signal processing in software on the main CPU. We measured time stolen by the same model of soft modem used by Jones et al. [41]. We agreed not to reveal the name of this modem, but it is based on an inexpensive chipset commonly found in commodity machines. Connecting the modem at 45333 bps (the highest our local lines would support) caused Windows 2000 + Rez to steal 9.87%$\pm$0.15 from a CPU reservation of 4 ms / 20 ms. We did not test the soft modem under Linux because driver support was not available.

Figure 11.5 showed that the percentage of CPU time stolen by the kernel while it processes incoming network data depends on the size of the reserved block of CPU time. This is because reserved time interferes with the user-level application that is causing data to arrive over the network.

There is no analogous user-level application for a soft modem: even when no data is being transferred the CPU must still process samples received over the phone line. Therefore, the proportion of time stolen by the soft modem driver does not depend on the size of reserved blocks of CPU time.

As Jones et al. mention, software-based implementations of Digital Subscriber Line (DSL) will require large amounts of CPU time: 25% or more of a 600 MHz Pentium III. Obviously a soft DSL driver will steal significant amounts of CPU time from applications if its signal processing is performed in a bottom-half context.

11.8.4 Serial and USB Ports

Neither Linux nor Windows 2000 showed any measurable amount of stolen time when receiving 115 Kbps of data over a serial port; probably the amount of incoming data is simply too small to cause problems. We did not test for stolen time while machines received data over the parallel port, but we hypothesize that there would be little or none since parallel port data rates are also small.

While retrieving a large file over USB (the file was stored on a CompactFlash memory card), a reservation of 4 ms / 20 ms under Windows 2000 + Rez had 5.7%$\pm$0.032 of its reservation stolen. USB could become a much more serious source of stolen time in the future as USB 2.0 becomes popular—it is 40 times faster than USB 1.1 (480 Mbps instead of 12 Mbps). Finally, while we did not test Firewire devices, at 400 Mbps it is a potentially serious source of stolen time.

11.9 Other Approaches and Related Work

11.9.1 Pessimistic Over-Reservation

The simplest way to ensure that applications with CPU reservations receive the full amount of processor time that they reserved would be to have the reservation subsystem pessimistically over-reserve. That is, to have it silently increase the amount of each reservation by a fixed factor (like we did in Figure 11.3). The factor would be the sum of the worst-case CPU utilizations of all devices that may steal time from applications—this would include most network, SCSI, IDE, USB, and firewire devices. This approach would have the advantage of not requiring the instrumentation of stolen time that is necessary for Rez-C and Rez-FB to work.

Over-reservation is a practical way of dealing with overheads that can be easily bounded, such as the overhead caused by timer interrupts. More sophisticated analysis of this type is also possible: Jeffay and Stone [35] showed how to guarantee schedulability for a dynamic-priority system when time is stolen by interrupt handlers, although the worst-case execution time and maximum arrival rates of interrupts needed be known. Unfortunately, bounding the amount of time that may be stolen by all devices in a desktop computer running a general-purpose operating system is not easy or straightforward.

Calculating the worst-case amount of CPU time that even a single device/driver combination can steal is difficult: the calculation must be performed for every hardware combination because it depends on processor speed, memory latency and bandwidth, and I/O bus speed. It must test every useful mode of a driver.[3] It also requires a test workload that correctly evokes worst-case response from the device driver; this will require multiple machines when testing network drivers. Furthermore, as we showed in Figure 11.5, for some drivers the amount of stolen time depends on the amount of the CPU reservation that they are stealing time from. We believe that these difficulties, multiplied by the hundreds of devices that must be supported by popular operating systems, imply that it is impractical to compute an accurate upper bound on the amount of time that bottom-half operating system processing may consume.

11.9.2 Moving Code into Scheduled Contexts

Systems programmers can manually reduce the amount of time stolen by a device driver by moving code from bottom-half contexts into scheduled contexts. For example, a DPC that moves a packet off of a network interface buffer could be converted into a very small DPC that awakens a worker thread that moves the packet. Jones et al. [41] have investigated the effects of moving CPU-intensive soft modem signal processing code from an interrupt handler first into a DPC, and then into thread context. In Section 11.8.3 we showed that a soft modem can steal close to 10% of a CPU reservation. In contrast, the amount of time stolen by the THR and RES versions of the soft modem driver described by Jones et al. would be small and most likely negligible.

The main problem with moving code out of time-stealing contexts is that to be successful overall, it requires fine-grained effort by a large number of driver developers, each of whom has an incentive to leave his or her code in a bottom-half context (that is more predictable, since bottom-half processing takes precedence over thread processing). This is an example of *priority inflation*, a social phenomenon that occurs when developers overestimate the priority at which their code should run. It happens because the motivations of driver developers (who want their device or subsystem to perform well) conflict with each other and with the motivations of system designers (who want overall predictability).

11.9.3 Scheduling Bottom-Half Activity

A number of recently designed operating systems can schedule device-driver activity. Because network interfaces are a principal source of asynchronous data and because TCP/IP stacks tend to involve copies and other CPU-intensive tasks, much of this work has focused on the scheduling of network protocol processing. Microkernels such as Mach [73] and Real-Time Mach [88] run device drivers and network protocols in user-level servers, allowing them to be scheduled normally. The Scout operating system [64] provides support for *paths*, channels through a layered communication system that can be explicitly scheduled. Nemesis [49] is a vertically-structured operating system that implements as much network processing as possible in shared libraries (rather than in shared servers or in a monolithic network stack) in order to simplify accounting and reduce crosstalk between applications. From our point of view the problem with these approaches is that most people are not interested in running a new OS: it is preferable to incrementally change an existing general-purpose OS, maintaining backwards application compatibility and approximately the same performance characteristics.

[3] For example, some of the options that can be turned on or off in the Linux IDE driver are: DMA, filesystem readahead, drive read lookahead, 32-bit I/O, drive defect management, multiple sector mode, and drive write-caching. All of these potentially affect the CPU usage of the driver.

Relatively little work has been done on scheduling bottom-half activity in general-purpose operating systems. Druschel and Banga [19] have implemented scheduling and proper accounting (but not real-time scheduling) of network processing in SunOS to solve the problem of *receiver livelock*: the situation in which all CPU time is used by low-level network receive processing, preventing applications from receiving packets at all. Our work does not address this problem.

Jeffay et al. [34] have modified the scheduling and network protocol processing in FreeBSD to provide integrated real-time application and protocol scheduling. They implemented a number of specializations that made the network subsystem fair and predictable, even in the presence of flooding attacks. However, instead of preemptively scheduling bottom-half activity they rely on the fact that the particular bottom-half primitives that they schedule run for a known, bounded amount of time.

Although both Rez-C and Rez-FB increase the predictability of CPU reservations, neither of them allows us to give any absolute guarantee to applications because there is no upper bound on the amount of time that Windows 2000 spends in bottom-half contexts. For example, Jones and Regehr [38] describe a Windows NT device driver that would unexpectedly enqueue a DPC that took 18 ms to run. To bound this time would require scheduling bottom-half time, rather than just accounting for it.

Making DPCs in Windows 2000 or bottom-half handlers in Linux preemptible would undoubtedly add some overhead: it would be equivalent to giving DPCs (which are currently very lightweight) a thread context. The median thread context switch time (with no address space switch) on our test hardware running an unmodified copy of Windows 2000 is $8.2\,\mu$s. While our test machine received data over 100 Mbps Ethernet it executed over 12,000 DPCs per second. If each DPC resulted in two thread context switches (one to a DPC thread and one back to the original thread), then roughly 20% of the CPU would be used just to perform the context switches. However, the reality is not likely to be this bad: DPCs could be batched, amortizing the context switch time, and a clever implementation of preemptible DPCs could probably run short DPCs without a thread context, only elevating them to full thread status in the (presumably uncommon) case where a DPC needs to be preempted.

A good implementation of scheduled DPCs might reduce system throughput only slightly on consumer systems that do not move large amounts of data around. However, both Linux and Windows 2000 are commonly used as servers that are connected to many hard disks and several high-speed network interfaces. In these situations adding overhead to critical device driver code paths is not acceptable: high system throughput is critical. Therefore, it seems unlikely that mainstream versions of Windows 2000 or Linux will preemptively schedule DPCs and bottom-half handlers in the near future. Perhaps future consumer versions of these systems will do so.[4]

Finally, even if DPCs are scheduled interrupt handlers will still steal time from applications. Because interrupts are scheduled in hardware using a static-priority scheduler, it is not possible to schedule them in software using a different algorithm. The best way to prevent time stealing by interrupt handlers from becoming a serious obstacle to predictability is to ensure, through testing and code inspection, that device drivers perform as little work as possible in interrupt context.

[4] "Forking" a popular OS like Windows 2000 or Linux into two versions may not be as bad of an idea as it first sounds if the difference between versions is strictly internal. In other words, if source and binary compatibility could be maintained at both the application and device-driver levels.

11.9.4 CPU Reservations

The Rez algorithm itself is not novel: it is very similar to the *constant utilization server* developed by Deng et al. [18], the *constant bandwidth server* that Abeni and Buttazzo developed [2], and the Atropos scheduler developed for the Nemesis OS [49]. However, we believe the augmented extensions of Rez—Rez-C and Rez-FB—to be novel: previous implementations of CPU reservations in general-purpose operating systems have not addressed the problem of time stealing by bottom-half processing.

Although Rez and similar algorithms use budgets and EDF internally, many other implementations of CPU reservations are possible. These include the dynamic creation of complete schedules in Spring [81], the tree-based scheduling plan in Rialto [40] and Rialto/NT [37], and rate-monotonic scheduling in the Fixed-Priority Open Environment [45].

11.9.5 Scheduling Based on Feedback and Progress

Rez-C and Rez-FB track an impediment to application progress, stolen time, and counteract it. A more aggressive approach would be to track application progress directly; it would then be possible to counteract all impediments to progress, not just the ones that we happen to measure. These include cache-related slowdown, memory cycle stealing by DMA, and actual changes in application requirements.

Steere et al. [83] track application progress by giving a reservation scheduler access to the state of buffer queues between the stages of pipeline-structured applications. Lu et al. [56] have used feedback to attain small numbers of missed deadlines at high system utilization when task execution times are not known. Finally, a number of models such as FARA by Roşu et al. [75] have used feedback about system load to select among different modes of operation for adaptive real-time applications.

11.10 Conclusions

This chapter has presented and evaluated two novel schedulers that help provide increased predictability to applications even when low-level operating system activity “steals” time from them. Our conclusion is that augmented CPU reservations are a useful, lightweight mechanism for increasing the predictability of time-based schedulers.

Chapter 12

Related Work

Chapter 2 presented a general survey of multimedia schedulers, applications, and programming models. The purpose of this section, on the other hand, is to explicitly compare elements of HLS with related work that has been described in the literature.

12.1 Hierarchical Scheduling

Systems providing hierarchical CPU scheduling can be divided into three broad categories. *Homogeneous* hierarchical schedulers use the same scheduling algorithm throughout the hierarchy. *General heterogeneous* hierarchies, such as HLS, allow user-specified scheduling algorithms everywhere in the hierarchy. In between these two extremes are *fixed heterogeneous* scheduling hierarchies that specify some of the schedulers in the hierarchy (usually including the root scheduler), possibly allowing user-specified schedulers at other locations in the hierarchy. Most of the existing hierarchical scheduling work falls into the last category.

12.1.1 Homogeneous Hierarchical Scheduling

Waldspurger and Weihl developed the *currency* abstraction in the context of *lottery scheduling* [90], a randomized proportional share scheduler. Currencies allow hierarchical isolation between groups of threads by providing a level of indirection between requests for scheduling and the actual number of shares that are allocated. Currencies are also supported by the *stride* scheduler [91], a deterministic proportional share scheduler based on virtual times.

The *borrowed virtual time* (BVT) scheduler [20] is a proportional share scheduler that allows selected threads to borrow against their future processor allocation, decreasing their scheduling latencies (at the expense of threads that are not allowed to borrow, or that can borrow less). A hierarchical version of BVT was proposed in order to reserve part of the processor bandwidth for multimedia applications, scheduling time-sharing applications in a best-effort manner using a second-level scheduler.

The main contribution of homogeneous hierarchical schedulers is that they provide hierarchical isolation between groups of resource consumers. They do not address the need for different kinds of scheduling support for applications with diverse requirements. Furthermore, the stride, lottery, and BVT schedulers do not address the problem of providing bounded-latency scheduling to real-time applications.

12.1.2 General Heterogeneous Hierarchical Scheduling

CPU inheritance scheduling [24] permits any thread to act as a scheduler by *donating* the CPU to other threads. The root of the scheduling hierarchy is a special thread that the operating system always donates the CPU to after the occurrence of a scheduling event. The scheduling hierarchy, then, exists as a set of informal relationships between threads, requiring very little support from the kernel.

CPU inheritance scheduling and HLS are equivalent in the sense that a scheduling hierarchy that can be expressed in one can be expressed in the other. However, HLS implements schedulers in the kernel in order to avoid causing unnecessary context switches. For example, on a 500 MHz PC running Windows 2000 a context switch between threads (the mechanism used for communication between schedulers in CPU inheritance) costs around 7 μs and a virtual function call (the mechanism used for communication between schedulers in HLS) costs about 20 ns. The other main difference between the two systems is that HLS was developed along with guarantees, a mechanism for reasoning about scheduler composition in order to provide guarantees to real-time applications, while CPU inheritance stipulates that all composition issues are the responsibility of the user.

As far as we know, HLS is the first general, heterogeneous hierarchical scheduling system to be implemented in a general-purpose operating system. CPU inheritance scheduling was prototyped in a user-level thread package and also implemented as part of the OSKit [22]. The Bossa domain-specific language (DSL) for application-specific scheduling policies [6] is being implemented in Linux and appears to support a general, heterogeneous scheduling hierarchy. Bossa represents an approach to making it easier to write schedulers that is more aggressive than the one taken by HLS: the DSL is used to guarantee safety properties of schedulers and also to specialize schedulers to meet the needs of specific applications.

12.1.3 Fixed Heterogeneous Hierarchical Scheduling

Scheduler activations [3] allow the OS to notify user-level thread schedulers of operating system events that may affect scheduling decisions, such as blocking and unblocking threads and granting and revocation of processors. The *virtual processor* interface in HLS is a generalization of scheduler activations; the relationship between the two is discussed in detail in Section 4.3.1.4. Nemesis [49] permits two-level scheduling hierarchies using an interface similar to scheduler activations. Both of these systems differ from HLS in that they require the second-level scheduler to execute in user space, they limit the scheduling hierarchy to two levels, and they fix the scheduler at the root of the hierarchy.

The Spin operating system [9] provides functionality similar to scheduler activations, but allows applications to load their own schedulers and thread package implementations into the kernel at run time. Therefore, Spin is more similar to HLS than scheduler activations are, although significant differences remain—Spin has a fixed root scheduler, and supports only two-level scheduling hierarchies. Vassal [14] also allows an application-specified scheduler to be loaded into the OS kernel at run time. Vassal is similar to HLS in that the loaded scheduler may take precedence over the time-sharing scheduler, but Vassal only allows a single scheduler to be loaded at a time, and consequently does not have to deal with the problem of schedulers whose demands for CPU time conflict. Also, in contrast with HLS, Vassal does not provide the loaded scheduler with the full set of notifications about OS events (such as thread blocking and unblocking) that a scheduler needs to be informed of.

The Exokernel [21] is an extensible operating system that uses a CPU donation primitive similar to the one used in CPU inheritance scheduling, allowing unprivileged applications to act as schedulers. However, the root scheduler for the Exokernel is fixed, and is not a real-time scheduler.

Resource containers [5] permit limits on various resources, including processor time, to be applied to hierarchical collections of threads. However, fair scheduling is guaranteed only on long time scales (tens of seconds). Share II [10] and the Solaris Resource Manager [58, Ch. 7] are commercial products that provide hierarchical fair-share scheduling, also on long time scales. Similarly, many general-purpose operating systems provide mechanisms outside of the main time-sharing scheduler for enforcing long-term limits on CPU usage. For example, the SIGXCPU signal in Unix operating systems [26, pp. 201–202] and job objects in Windows 2000 [77, pp. 374–378] can be used to notify or terminate processes or groups of processes that have exceeded their CPU usage limits. All of these mechanisms were designed to achieve long-term fairness among non-real-time applications—they are not suitable for real-time scheduling.

Goyal et al. invented *start-time fair queuing* (SFQ) [28], a proportional share scheduling algorithm that works well in hierarchical environments. They proposed an architecture that uses SFQ schedulers at all points in the scheduling hierarchy except for leaves, which may implement arbitrary scheduling algorithms such as rate monotonic scheduling or earliest deadline first. However, no method for guaranteeing the schedulability of applications running under the leaf schedulers was presented. Also, the fact that the guaranteed scheduling latency of SFQ depends on the number of threads being scheduled by a SFQ scheduler makes it a dubious choice for scheduling real-time application that have deadlines of the same order of magnitude as the length of the system scheduling quantum (as Section 5.3.3 showed).

RED-Linux [92] defines a two-level scheduling framework in which each task is characterized by a 4-tuple consisting of priority, start time, finish time, and budget. Multiple leaf schedulers may exist, and each task must belong to one of them. A leaf scheduler is a function from a task's scheduling parameters to an *effective priority*; the root scheduler always runs the task that has the highest effective priority. This framework is general enough to implement many existing classes of real-time schedulers. However, it supports no mechanism for resolving conflicts among leaf schedulers, and consequently cannot provide throughput or delay guarantees to applications.

The *open environment for real-time applications* developed by Deng and Liu [18] and the BSS-I [51] and PShED [52] frameworks developed by Lipari et al. are hierarchical scheduling systems designed around the idea of providing a *uniformly slower processor* (USP) to each real-time application. The USP abstraction guarantees that any task set (i.e. the set of real-time tasks comprising a real-time application) that can be scheduled without missing any deadlines (by a particular scheduler) on a processor of speed s can also be scheduled by that scheduler if it is given a USP with rate s/f on a processor of speed f.

The distinguishing feature of the USP guarantee is that it is characterized only by a rate (a share of a processor) and not by a granularity at which that rate will be granted. Granularity information must be specified dynamically: a scheduler receiving a USP guarantee must inform the root scheduler of every deadline that matters to it. The root scheduler then performs a computation over the deadlines provided by all of its children in order to ensure that no USP uses more than its share of the processor over any time interval that matters to any USP. Therefore, USPs implicitly require all deadlines to be known at run time.

The requirement that all deadlines be specified dynamically adds run-time overhead and is not a good match for some kinds of schedulers. For example, a rate monotonic scheduler retains no information about task deadlines at run time, and consequently cannot make use of a USP guarantee. Similarly, proportional share schedulers and time-sharing schedulers do not have explicit deadlines. Furthermore, the BSS-I and PShED schedulers entail considerable implementation complexity, requiring a balanced-tree data structure if run-time cost linear in the number of active deadlines is to be avoided. The USP guarantee fits into the HLS framework as a special case of the CPU reservation abstraction. However, its restricted programming model—deadlines must be known at run time—limits the utility of the USP approach for scheduling unmodified real-time applications in a general-purpose operating system.

12.2 Resource Management for the CPU

The HLS resource manager differs from all previous CPU resource managers in one important respect: it has available to it a collection of hierarchical schedulers, as well as reflective information about those schedulers, that allows it to match each application request with a scheduler that can meet its requirements. When appropriate, it provides applications with guarantees about the rate and granularity with which they will receive CPU time.

Another important distinction is that many of the resource managers that have appeared in the literature are explicitly designed to support either adaptive applications, networked applications, or both. Although the HLS resource manager could be used to support these kinds of applications, they are not its main target. Distributed applications are not a principal concern by assumption: the HLS resource manager is designed to solve problems associated with the allocation of local processor time. Adaptive applications are not a principal concern because the vast majority of existing multimedia applications are not resource self-aware, and do not attempt to dynamically adjust their behavior in response to changing resource availability.

The QoS Broker [65] is a middleware resource manager that reserves network resources (bandwidth, buffer space, etc.) and operating system resources (real-time scheduling capacity, memory, storage, etc.) using information stored in *profiles*, in order to meet applications' needs. The goals of the HLS resource manager and the QoS broker overlap to some extent—both are designed to store resource information for applications and then retrieve and apply it when needed—but the similarities appear to end there because the QoS broker focuses primarily on the management of network resources and the HLS resource manager focuses primarily on CPU time.

The Dynamic QoS Resource Manager (DQM) [12, 13] is designed to operate without sophisticated scheduling support from the operating system. Real-time applications are expected to support a number of different *execution levels*, each of which is characterized by a 3-tuple consisting of *resource usage*, *benefit*, and *period*. The DQM then selects an execution level for each application such that high overall benefit is provided to the user. It reduces overall resource usage when applications miss deadlines and increases usage when idle time is detected. Though there are similarities between the HLS resource manager and the DQM (i.e. applications are characterized by the resources they require), the approaches are otherwise very different. DQM was designed under the assumption that applications may be freely modified, and that the services provided by existing general-purpose operating systems are sufficient to schedule multimedia applications. HLS, on the other hand, was designed under the assumption that unmodified multimedia applications must be supported, and that the operating system may be freely modified in order to implement diverse scheduling algorithms. The justification for the HLS approach is that since there are many applications it is unacceptable to complicate the application programming model.

Oparah [70] describes a QoS manager (QM) for the Nemesis operating system that is similar in many ways to the DQM: it requires applications to register different *modes* with the OS, with each mode being characterized by a *resource tuple* and a *value*. The QM then attempts to provide high overall value during overload by reducing the resource usage of some or all applications. A novel feature of the QM is a graphical interface that allows users to indicate special preferences such as forbidding the system from taking resources from an important application or correcting suboptimal resource allocations; the QM remembers these corrections and attempts to track user preferences more closely in the future. Like the DQM, the QM focuses on adaptive applications.

The modular resource manager for Rialto [36] divides the system into *resource providers* and *activities*. Each activity requests a *resource set*, a list of amounts of the different resources it requires to operate correctly, from a central planner. During overload, user preferences are consulted to resolve the conflict. The Rialto and HLS resource managers are similar in intent, but differ in a number of details. First, and most importantly, Rialto assumes that applications are resource-self aware, meaning that they are able to make good use of amounts of resources other than their full requirements. We reject this assumption—some applications are simply incapable of functioning correctly when they receive less CPU time than they require, and others are capable of adapting in principle, but have not been implemented in a manner that makes this possible. Second, the Rialto resource manager was designed to allocate multiple resources while the HLS resource manager is only concerned with the CPU. Finally, the Rialto resource manager assumes that resource amounts may be combined linearly, meaning that a resource with overall capacity 1.0 will always be able to support two activities that each require 0.5 of the resource. This assumption is not true for CPU time, for example, when the rate monotonic scheduling algorithm is used, or when schedulability analysis includes elements such as context switch overhead or blocking factors. The HLS resource manager uses scheduler-specific schedulability analysis routines to determine whether a particular set of demands for processing time is feasible or not.

Chapter 13

Conclusions

This chapter concludes the dissertation by summarizing its contributions and discussing some possible avenues for future work.

13.1 Summary of Contributions

HLS is the first general, heterogeneous hierarchical scheduling system to be implemented in a general-purpose operating system. The thesis that this dissertation has supported is that *extending a general-purpose operating system with general, heterogeneous hierarchical scheduling is feasible and useful*.

The feasibility of HLS was demonstrated by presenting a design of the hierarchical scheduler infrastructure that is based on a novel extension of the virtual processor abstraction that was developed for the work on *scheduler activations* [3], and also by implementing it in the Windows 2000 kernel. Basic mode-change operations such as moving a thread between schedulers, creating and destroying a scheduler, and beginning and ending a CPU reservation were shown to be fast, taking less than $40\,\mu$s on a 500 MHz Pentium III. We also showed that although HLS increases the cost of a context switch slightly, the performance penalty caused by a context switch to a thread in terms of re-establishing its working set in the processor cache can easily be two orders of magnitude greater than the cost added by HLS.

The usefulness of HLS was established in the following ways. First, by surveying a number of scheduling behaviors that have been developed to support multimedia applications, and the diverse requirements of multimedia and other soft real-time applications, it was concluded that at least three types of scheduling behavior are useful: time-sharing scheduling for batch and interactive applications, CPU reservations with a precise granularity and minimum amount to support real-time applications whose value does not degrade gracefully, and best-effort real-time scheduling for applications whose value degrades gracefully. Second, using three application scenarios it was shown that different ways of using a general-purpose operating system can benefit from distinct types of scheduling. Third, it was shown that the guarantees provided by a broad class of multimedia scheduling algorithms can be reasoned about in a formal way, and a novel method was presented for verifying that scheduling hierarchies are guaranteed to be correct in the sense that application threads receive guaranteed scheduling behavior. Fourth, it was shown that a number of complex, idiomatic scheduling behaviors can be composed using small hierarchical schedulers as components. Fifth, the design of a user-level resource manager was presented in order to show that a resource manager can make good use of the flexible scheduling provided by HLS to enforce high-level policies about the allocation of CPU time. Finally, two novel schedulers were presented that can help increase application predictability when a general-purpose operating system steals CPU time from a running application.

The value provided by HLS is that it allows a computer running a general-purpose operating system to be used in situations where the inflexibility of the CPU scheduler had previously prevented it from being used. Rather than statically adding scheduling behaviors to the operating system, we have created a dynamic architecture for extending the scheduling subsystem, allowing future improvements in scheduling algorithms to be easily incorporated into the OS.

13.2 Future Work

13.2.1 Scheduler Composition

A useful area of further research in scheduler composition would be, for each existing real-time scheduler, to find the (provably) weakest guarantee that allows it to still provide the desired guarantees to its children. To do this it may be necessary to develop guarantee types other than the ones described in Chapter 5, that are a better match for the scheduling semantics provided by particular real-time schedulers. It would be interesting to investigate the theoretical and practical differences between "static" guarantees like CPU reservations, whose scheduling behavior requires no information other than rate and granularity, and "dynamic" guarantees such as the uniformly slower processor, that require fine-grained information about deadlines in order to be useful.

There are many global computations over the scheduling hierarchy and set of running applications that would be desirable to perform. For example, when worst-case semaphore hold times are known, global schedulability analysis could be performed in the presence of synchronizing tasks. Also, it would be useful to be able to test arbitrary scheduling hierarchies for desirable global properties such as good processor affinity behavior and low overall expected context switches.

13.2.2 Hierarchical Scheduler Infrastructure

Since the HSI is a very general tool for building scheduling behaviors, it would be desirable to release it to other research groups, who could then spend more time developing and using interesting schedulers and less time understanding the internals of general-purpose operating systems or running schedulers in simulated operating systems.

A useful extension to the HSI would be to design a version that is compatible with medium- and large-sized multiprocessor machines where it is infeasible to serialize scheduling decisions across all processors. A straightforward space-sharing approach would be to have different scheduling hierarchies control processor allocation on disjoint subsets of the processor pool. Although scheduler operations within each hierarchy would be serialized, no synchronization across hierarchies would be required. To migrate a thread between hierarchies, it would suffice to unregister a thread from one hierarchy and register it with another. Extending the HLS scheduling model to support relaxed synchronization within a single scheduling hierarchy would be a more difficult task.

Although HLS lowers the barrier for implementing a new scheduler, writing a scheduler is still more difficult than it seems like it should be. It would be interesting to develop a restricted *domain specific language* for schedulers that, along with run-time support from the HSI, (1) ensures that each scheduler invocation terminates in a bounded (and short) amount of time, (2) prevents schedulers from leaking resources, (3) is type-safe, preventing schedulers from reading or writing forbidden memory areas, (4) ensures that schedulers obey the HLS protocol, (5) ensures that schedulers cannot divide by zero, generate a page fault, corrupt processor floating point state, or perform other destructive actions, and (6) validates or automates the tedious and error-prone updates to the states of virtual processors, ready queues, and other data structures. Perhaps model checking techniques, in addition to language- and system-based techniques, could be used to verify some of these properties, at least for schedulers with a small state space. The "holy grail" for a domain specific scheduler language would be for it to ensure that the scheduler actually provides the guarantees that it promises to provide. This, in combination with the properties listed above, could lead to provable scheduling behavior under weak assumptions (i.e., that the hardware is not faulty) rather than the strong assumptions that were presented in Section 5.1.3. The Bossa framework [6] is addressing a number of these issues using a domain-specific language for writing schedulers.

The current implementation of the hierarchical scheduler infrastructure is a prototype, and there are many ways in which it could be improved. It should be profiled and optimized, in order to narrow or close the gap between the context switch time of the native Windows 2000 scheduler and a context switch within a single, well-implemented HLS scheduler. A solution to the problem of priority inversion would be a desirable feature for the HSI, as would a mechanism permitting schedulers to send notifications to user-level applications. Finally, a precisely settable interrupt mechanism would make Windows 2000 into a more viable real-time operating system for applications with deadlines on the order of a small number of milliseconds.

13.2.3 Resource Manager

The obvious future work for the resource manager is to implement it. Beyond that, there are fertile grounds for future work in the areas of parameter estimation for real-time applications, especially those with data-dependent CPU requirements, and GUI-based resource control interfaces that end-users can understand and manipulate. Also, it would be useful to develop a better understanding of how to tune the degree of pessimism in resource reservations for applications in order to balance the requirements of high overall utilization and low deadline misses. Finally, it would be interesting and useful to develop rules for the resource manager that implement a number of complex scheduling behaviors, and to ensure that these complex behaviors compose correctly.

13.2.4 Augmented Reservations

The augmented reservation schedulers can be viewed as a necessary evil: if general-purpose operating systems did not have the bad behavior of stealing time, augmented reservations would not be necessary. A useful project would be to move the Windows 2000 DPC mechanism into thread context, eliminating this source of stolen time. This project is ambitious because it would be difficult to avoid reducing the throughput of DPC-intensive workloads, and because scheduling DPCs begs the question of what their scheduling parameters are. In other words, if DPC threads are scheduled at the highest priority, then there is no advantage to scheduling them as threads. If they do not have the highest priority, then we must know for how long their execution can be delayed.

Appendix A

Glossary

Definitions are collected here for easy reference. In general, the accepted definitions for terms are used, although some terms are used in a more restricted sense than their usual interpretation. For example, *scheduler* always refers to a CPU scheduler.

adaptive application A broad term for applications that can adjust their resource usage to compensate for variations in the level of service that they receive. Adaptation can be on a long time scale, for example, if an application can make use of different guarantees, or on a short time scale if an application sheds load in response to a transient shortage of a resource (for example, a video player application can drop frames to “catch up” after being starved).

admission control A first-come first-serve resource allocation policy in which requests for guarantees are rejected if the resulting total utilization would exceed some threshold.

application A user-level program designed to serve a specific purpose. Although applications may be implemented by multiple processes, in this dissertation it may be assumed that an application is implemented by a single process.

best effort A resource allocation policy in which no request for scheduling is rejected. Consequently, no level of service is guaranteed.

bottom-half activity Low-level operating system activity that has higher priority than, and is invisible to, the thread scheduler of a general-purpose operating system.

closed system Describes a way of using an operating system characterized by knowledge of the set of applications that will be run, and their relative importances. This permits a priori schedulability analysis.

deadline A time, usually externally imposed, by which a real-time computation must complete.

dispatch latency The time between when a thread is scheduled and when it begins to execute. Theoretically, in a preemptive OS the dispatch latency for a high-priority thread should be very low. However, in practice preemptive OSs are non-preemptive at times; for example, while running an interrupt handler. The duration of the longest possible non-preemptive interval is said to be the worst-case dispatch latency of an OS.

earliest deadline first (EDF) A scheduling algorithm that always runs the task with the earliest deadline. EDF is optimal in the sense that if any algorithm is able to schedule a task set, EDF will be able to schedule it.

enforcement The mechanism by which isolation and guarantees are implemented. By enforcing limits on the CPU utilization of one application, processor time can be guaranteed to other applications.

general-purpose operating system (GPOS) An operating system designed to meet a variety of goals, including protection between users and applications, fast response time for interactive applications, high throughput for batch and server applications, and high overall resource utilization. Unix and Unix-like operating systems are general-purpose operating systems, as are members of the Microsoft Windows family.

graceful degradation Characterizes an application whose utility decreases smoothly, rather than sharply, when it receives insufficient CPU time. The opposite is non-graceful degradation, which characterizes applications that produce little or no value when their full requirements are not met.

guarantee In general, a *contract* between the operating system and a resource consumer (thread, application, user, etc.) promising that the consumer will receive a guaranteed level of service with respect to some resource for the duration of the reservation. In this dissertation the resource being guaranteed is always CPU time. Guarantees are considered to be long-lived entities (i.e. lasting for seconds, minutes, or longer) when compared to the amount of time between scheduling decisions (i.e. milliseconds).

hard real-time system Characterization of a system where meeting application deadlines is the primary metric for success.

hard real-time task A task that loses all value (or generates negative value) if it is not completed by its deadline.

hierarchical scheduling A generalization of scheduling in which schedulable entities may themselves be schedulers.

HLS Abbreviation for Hierarchical Loadable Schedulers. A general term for the scheduling architecture presented in this dissertation.

HSI Abbreviation for hierarchical scheduler infrastructure. A software library implemented as part of an operating system kernel that allows a hierarchy of schedulers to allocate CPU time. An important part of the HLS architecture.

isolation An application is isolated from other applications if it is guaranteed to receive a certain level of service regardless of the behavior of other applications.

loadable scheduler An implementation of a scheduler that can be dynamically loaded into an operating system kernel, where instances of it may become part of a scheduling hierarchy.

monolithic scheduler A scheduling algorithm that is not implemented in a hierarchical way. Although monolithic schedulers lack flexibility, they are equivalent to hierarchical schedulers in the sense that every scheduling hierarchy could also be implemented as a monolithic scheduler.

multimedia operating system An operating system that supports multimedia applications. The connotation is that applications are soft real-time and that the system is an open system. Most multimedia operating systems are general-purpose operating systems that have been extended to better support soft real-time applications. A few operating systems, such as BeOS, were specifically designed to support multimedia.

multimedia Literally, the class of applications that brings together different media; for example, sound, still images, video, and text. Colloquially (and in this dissertation) multimedia refers to applications that perform ongoing computations to handle audio, video, or other streaming data. Multimedia applications are often soft real-time applications.

multi-threaded application An application that is structured as a group of cooperating threads. For example, a video player application might have a thread for each of the following jobs: reading data from disk, decoding data, displaying frames of decoded data to the screen, and updating the user interface.

non-preemptive scheduler A scheduler that may switch contexts only when a thread explicitly yields the processor.

open system Describes a way of using an operating system that is characterized by a lack of advance knowledge about either the set of applications that will be run or their relative importances to the user or users. General-purpose operating systems are often used as open systems, meaning that users are free to install and run a new application at any time, with the expectation that all running applications will work properly as long as no resource (for example, memory, disk bandwidth, or CPU bandwidth) is oversubscribed.

Pentium timestamp counter A counter present on Pentium-class x86 processors that stores the number of cycles since the machine was turned on, as a 64-bit integer. It can be efficiently read using the `rdtsc` instruction. On a 500 MHz machine, for example, the timestamp counter has a resolution of 2 ns.

period The rate at which a real-time application requires computations to be performed. In the general case deadlines do not need to be equal to period boundaries. However, for multimedia applications it is usually the case that they are.

preemptible operating system An operating system that may preempt a thread even when it is executing in the kernel. Solaris, BeOS, and Windows 2000 are preemptible, but Linux, FreeBSD, and Windows 95/98/Me are not.

preemptive operating system An operating system whose scheduler is preemptive. Nearly all general-purpose operating systems are preemptive (as of version 9, the Macintosh OS is not preemptive).

preemptive scheduler A scheduler that may switch between threads at any time.

priority inheritance An algorithm that eliminates some classes of priority inversions [76].

priority inversion The condition in which a high-priority thread is kept from running by a low-priority thread. For example, priority inversion occurs when a low-priority thread is executing inside of a critical section and a high-priority thread is waiting to enter the critical section. If a medium-priority thread preempts the low-priority thread (while still inside of the critical section), the result is *unbounded priority inversion*—the low- and medium-priority threads can indefinitely prevent the high-priority thread from running.

processor affinity A scheduling heuristic that attempts to keep from moving threads between processors on a multiprocessor to avoid incurring the cost of moving the thread's working set between processor caches.

rate monotonic (RM) A scheduling algorithm that assigns priorities to periodic tasks in order of increasing period length. RM is optimal among static-priority scheduling algorithms in the sense that if any static-priority scheduler can schedule a task set without missing deadlines, then RM can also.

real-time operating system An operating system designed to support real-time tasks. The connotation is that the applications are hard real-time and that the system is not an open system.

real-time The class of computations whose correctness depends not only on whether the result is the correct one, but also on the time at which the result is delivered. Real-time applications are those that perform any real-time computations.

resource management A broad term, most often used to describe resource allocation policies more sophisticated than admission control or best effort.

resource manager A software entity that performs resource management.

scheduler Either an algorithm or an implementation of an algorithm that multiplexes a resource among different entities that all require access to the resource. In this dissertation *scheduler* and *CPU scheduler* are used synonymously.

schedulability analysis The process of deciding whether or not a scheduler can schedule a particular set of tasks without missing any deadlines. Schedulability analysis is often quite simple; for example, in the most restricted case the schedulability of a task set by an EDF scheduler may be performed by summing up the utilizations of all tasks: if the total is less than or equal to one the task set is feasible, otherwise it is infeasible. Schedulability analysis for complex task sets, particularly on multiprocessors, can be difficult or intractable.

scheduling hierarchy A tree (or directed acyclic graph) of schedulers. Besides the root scheduler, every scheduler in the hierarchy has one (and occasionally, more than one) *parent*—the scheduler(s) adjacent to the scheduler in the direction of the root of the scheduling hierarchy. Each scheduler that is not a *leaf scheduler* has one or more *child* schedulers—schedulers that are adjacent in the direction away from the root.

server call A remote procedure call (RPC) based mechanism for providing an operating system service to a thread.

soft real-time system Characterization of a system designed to support soft real-time tasks. The implication is that other system design goals (such as achieving high average throughput) may be as important, or more important, than meeting application deadlines.

soft real-time task A task that has no fixed deadline or that retains some value after its deadline has passed. For example, interactive applications are soft real-time: the user may become increasingly irritated as the application fails to respond, but there is no clearly defined time at which the application loses all value.

stolen time Time that is "stolen" from a user-level application by bottom-half operating system activity such as interrupt handlers or deferred procedure calls. The time is stolen because, in most general-purpose operating systems, it is invisible to the scheduler and therefore not accounted for.

system call A coroutine-based mechanism for providing an operating system service to a thread. In a preemptible operating system it is irrelevant to the scheduler whether a thread is executing a system call or not.

task A single instance of a real-time computation that must complete by a certain time. A real-time application may be comprised of several threads, each of which may perform many tasks. Also, in the context of real-time scheduling theory, "task" is often used to refer to an entity with real-time requirements.

thread A flow of control. *Kernel threads* are supported by the operating system, while *user-level threads* are implemented outside of the kernel. In this dissertation all threads are assumed to be kernel threads.

usage scenario The way a particular computer is used, viewed at a high level. Example usage scenarios include: supporting interactive and multimedia applications for a single user, serving web pages, or supporting interactive applications for more than one user.

utility The subjective benefit generated by an application, running at a particular time, to a particular user. Synonymous with *value*.

utilization The fraction of a resource's total capacity that is being used.

Appendix B

The Loadable Scheduler Interface

This appendix builds upon material presented in Chapters 4 and 9. It describes the programming interface provided by the Hierarchical Scheduler Infrastructure (HSI) for implementing loadable schedulers.

B.1 Data Structures

B.1.1 Scheduler Instance

The *scheduler instance* data structure is used by the HSI to keep track of scheduler instances. Scheduler instance memory is managed by the HSI; the data structure is guaranteed to have been allocated by the time an instance's `I_Init` function is called, and will not be deallocated until after its `I_Deinit` function has returned. The fields of the scheduler instance are shown in Figure B.1. They are used as follows:

- `Name` — a character array containing a string that uniquely identifies the scheduler; this field is set by the HSI.
- `Type` — a character array containing a string indicating what guarantees the scheduler requires and provides (not currently used).
- `CB` — a pointer to a struct of type `HLS_CALLBACKS`; this field is set by the HSI at instantiation time.
- `SchedData` — The type of this field, `SDHandle`, is an alias for `void *`. This field can be used to point to a scheduler-specific variable. For example, the HLS time-sharing scheduler uses this field to point to a struct containing the ready queue and related data structures.
- `HLSTimer` and `HLSDpc` — these fields are for internal use by the HSI and must not be touched by schedulers.

B.1.2 Virtual Processor

Virtual processors embody parent / child relationships between two schedulers: each scheduler shares one or more virtual processors (VPs) with its parent(s) and with its children. The responsibility for correctly allocating and deallocating memory for VPs lies with the child. That is, before

```
struct HLS_SCHED_INSTANCE {
  char Name[MAX_NAMELEN];
  char Type[MAX_NAMELEN];
  struct HLS_CALLBACKS *CB;
  SDHandle SchedData;

  // private fields for use only by the HSI
  KTIMER HLSTimer;
  KDPC HLSDpc;
};
```

Figure B.1: Fields of the scheduler instance structure

registering a VP with its parent a child must first allocate and initialize it. The fields in a virtual processor, shown in Figure B.2, are:

- `Proc` — This field is always set either to the number of the physical processor that has been granted to the VP or to the constant `NO_PROC`, indicating that the VP is not currently running.

- `State` — The state of a VP is represented by an enumerated type that may take the value `VP_Ready`, `VP_Waiting`, or `VP_Running`.

- `TopSched` — A pointer to the scheduler instance that is the parent in a relationship represented by a VP.

- `BottomSched` — A pointer to the scheduler instance that is the child in a relationship represented by a VP.

- `UpperData` — The type of this field, `TDHandle`, is an alias for `void *`. `UpperData` may be used to store scheduler-specific per-VP data. For example, the HLS time-sharing scheduler uses `UpperData` to store a pointer to a structure containing the priority and ready queue entry for a VP.

- `TopData` and `BottomData` — These fields are intended to make writing a scheduler more convenient by allowing a top virtual processor to point to the bottom virtual processor that it is currently granting a processor to, and vice versa.

B.1.3 Scheduler Callbacks

Schedulers, like other device drivers, are event-driven. They must make a collection of function pointers available to the rest of the system in order to receive notifications when events of interest occur. This collection of functions is shown in Figure B.3; it also includes a single variable `Name`. The namespace of schedulers is separate from the namespace of scheduler instances; the HSI keeps track of both kinds of entities. The callbacks can be divided into three categories, identified by a prefix. First, callbacks that are initiated by a scheduler's child (or potential child) have the prefix "`B`". Callbacks from a parent have the prefix "`T`". Finally, callbacks from the HSI have the prefix

```
struct HLS_VPROC {
  PROCT Proc;
  VPROC_STATE State;
  struct HLS_SCHED_INSTANCE *TopSched;
  struct HLS_SCHED_INSTANCE *BottomSched;
  TDHandle UpperData;
  struct HLS_VPROC *BottomData;
  struct HLS_VPROC *TopData;
};
```

Figure B.2: Fields of the virtual processor structure

```
struct HLS_CALLBACKS {
  char *Name;

  // callbacks from children

  HLS_STATUS (*B_RegisterVP) (struct HLS_SCHED_INSTANCE *Parent,
                              struct HLS_SCHED_INSTANCE *Child,
                              struct HLS_VPROC *VP);
  void (*B_UnregisterVP) (struct HLS_VPROC *VP);
  void (*B_VP_Request) (struct HLS_VPROC *VP);
  void (*B_VP_Release) (struct HLS_VPROC *VP);
  HLS_STATUS (*B_Msg) (struct HLS_SCHED_INSTANCE *Inst,
                       struct HLS_VPROC *VP,
                       MsgHandle Msg);

  // callbacks from parents

  void (*T_VP_Grant) (struct HLS_VPROC *VP);
  void (*T_VP_Revoke) (struct HLS_VPROC *VP);

  // callbacks from HSI

  void (*I_Init) (struct HLS_SCHED_INSTANCE *Self,
                  struct HLS_SCHED_INSTANCE *Parent);
  void (*I_Deinit) (struct HLS_SCHED_INSTANCE *Self);
  void (*I_TimerCallback) (struct HLS_SCHED_INSTANCE *Self);
  void (*I_CheckInvar) (struct HLS_SCHED_INSTANCE *Self);
};
```

Figure B.3: Fields of the scheduler callback structure

"I". Most callbacks take a parameter called `Self`, which is analogous to the implicit self parameter passed to object methods in languages such as C++. The functions that each scheduler provides to the HSI are:

- `B_RegisterVP` — One scheduler makes this call to another to establish a parent / child relationship. If successful, `HLS_SUCCESS` should be returned, indicating that the calling scheduler has become a child of the called scheduler.

- `B_UnregisterVP` — This call, made by a child scheduler, ends a parent / child relationship between two schedulers.

- `B_VP_Request` — A child scheduler makes this callback to notify its parent that a waiting VP is now ready to be scheduled.

- `B_VP_Release` — A child scheduler makes this callback to notify its parent that a ready or running VP can no longer use a physical processor.

- `B_Msg` — This callback allows arbitrary messages to be sent to schedulers; the argument of type `MsgHandle` is an alias for `void *`. This function uses the "B" prefix because messages are often sent from a scheduler to its parent (for example, to change the parameters of a CPU reservation), but messages can also originate in the HSI.

- `T_VP_Grant` — A scheduler instance receives this callback when its parent allocates a physical processor to a virtual processor that is in the `VP_Ready` state.

- `T_VP_Revoke` — This callback notifies a scheduler that a physical processor is being revoked from a virtual processor that is currently in the `VP_Running` state.

- `I_Init` — This is guaranteed to be the first callback received by a scheduler instance. It should be used to initialize data structures in such a way that when `Init` returns, the scheduler is ready for VPs to register with it.

- `I_Deinit` — This callback is guaranteed not to occur while any VPs are registered with a scheduler instance. It gives an instance the opportunity to deallocate dynamic memory before being destroyed.

- `I_TimerCallback` — This function is called when a scheduler's timer expires, permitting it to make a scheduling decision.

- `I_CheckInvar` — This callback gives schedulers the opportunity to ensure that their internal data structures are consistent. It is acceptable to flag an inconsistency by printing a debugging message or by halting the system. `CheckInvar` will only be called when the hierarchy is in a "stable" state—when it is not in the middle of processing a notification.

B.2 Functions Available to Schedulers

The HSI makes a number of functions available to loadable schedulers; their prototypes are shown in Figure B.4.

```
void HLSSetTimer (WNT_TIME Time, struct HLS_SCHED_INSTANCE *Sched);

WNT_TIME HLSGetCurrentTime (void);

void *hls_malloc (int size);

void hls_free (void *mem);

void HLS_ASSERT (expr);

void HLSDbgPrint (level, (format, ...));
```

Figure B.4: Functions provided to scheduler instances by the HSI

B.2.1 `HLSSetTimer`

Schedulers may use this function to arrange for a timer callback to arrive in the future. Following Windows convention, negative times are relative and positive times are absolute (setting `Time` to 0 results in an immediate callback). Currently each scheduler gets only one timer; successive calls to `HLSSetTimer` result in the timer being reset. The resolution of this timer is tunable and depends on clock interrupt frequency. Currently, it can be assumed to be accurate to approximately 1 ms.

B.2.2 `HLSGetCurrentTime`

This function returns the current time in 100 ns units. The time is based on the value returned by the `rdtsc` instruction; this means that reading the time is cheap and precise, but it may not be accurately calibrated with respect to the times used by Windows 2000.

B.2.3 `hls_malloc`

Allocate a block of memory—the interface is the same as the C library function `malloc`.

B.2.4 `hls_free`

Release a block of memory—the interface is the same as the C library function `free`.

B.2.5 `HLS_ASSERT`

If `expr` is false, print the line number, file, and expression of the failing assert to the console. Also attempt to trap to the kernel debugger or, if the debugger is unavailable, halt the system.

B.2.6 `HLSDbgPrint`

This macro prints a debugging message to the console if `HLS_DBG_PRINT_LEVEL` is greater than or equal to the level parameter. The arguments `(format, ...)` are evaluated similarly to the arguments to the C library function `printf`.

B.3 Constants, Variables, and Types

B.3.1 `WNT_TIME`

This type is used by HLS to represent time values. It is an alias for the signed 64-bit integer.

B.3.2 `HLS_STATUS`

This enumerated type is used as a return value by many HLS functions. `HLS_SUCCESS` is the success value. Failure values include `HLS_NOROOM` and `HLS_INVALID_PARAMETER`.

B.3.3 `HLS_MAX_PROCS`

This integer constant indicates the most processors that a Windows 2000 system could contain. It will probably always be equal to 32.

B.3.4 `HLS_MAX_NAMELEN`

The maximum length, in bytes, of the name of a scheduler, scheduler type, or scheduler instance.

B.3.5 `HLSNumProcs`

This integer variable indicates the number of processors currently available in the system. It starts at 1 during boot and increases as more processors are brought online. Schedulers that load after boot time may treat it as a constant. Processors are always numbered 0..`HLSNumProcs`-1.

B.3.6 `HLSCurrentProc`

This integer variable is equal to the number of the processor on which scheduler code is currently executing.

B.3.7 `NO_PROC`

This is an integer constant that a scheduler can sometimes use when a processor number is required, indicating that no particular processor is preferred.

B.3.8 `HLS_DBG_PRINT_LEVEL`

This integer variable is used to control the amount of debugging information that appears on the console. A value of 0 causes HLS to be completely silent, while 9 is very verbose.

B.4 Functions for Loading and Unloading a Scheduler

The following two functions, unlike all other functions in this appendix, do not pertain to a particular scheduler instance. Rather, they are called by a scheduler device driver as it is being loaded into or unloaded from the Windows 2000 kernel. Schedulers that are built into the kernel should ignore these functions.

B.4.1 `HLS_STATUS HLSRegisterScheduler (struct HLS_CALLBACKS *)`

Every Windows 2000 device driver provides a `DriverEntry` callback that is called by the operating system after the driver has been loaded into the kernel address space. When a scheduler is loaded, its `DriverEntry` function should at some point call `HLSRegisterScheduler` to inform the HSI of its presence, passing the address of the scheduler's callback table (described in Section B.1.3) as a parameter.

B.4.2 `HLS_STATUS HLSUnregisterScheduler (struct HLS_CALLBACKS *)`

Prior to unloading a device driver, Windows 2000 calls the driver's `DriverUnload` function. When the driver being unloaded is an HLS scheduler, it should call `HLSUnregisterScheduler` to notify the HSI of its absence.

Appendix C

An Example Scheduler

This appendix contains complete, working source code for the HLS proportional share scheduler. Its purpose is to provide a concrete example of code using the scheduling primitives presented in Chapters 4 and 9, and in Appendix B.

```
/*++

Copyright (c) 2000, 2001 Dept. of Computer Science, University of Virginia

Module Name:

    hls_sched_ps.c

Abstract:

    Uniprocessor proportional share scheduler, based on SFQ and BVT.

Author:

    John Regehr 14-Feb-2000

Environment:

Revision History:

--*/

/*
 * static per-instance data must live in this struct
 */
struct PS_INSTANCE_DATA {
  struct HLS_VPROC TVP;
  int PSInitState;
  SCHAR DefaultPri;
  struct HLS_SCHED_INSTANCE *Self;
  int NumThreads;
  LIST_ENTRY AllThreads;
  ULONG TimeIncrement;
};
```

```
/*
 * per-bottom-VPROC data lives in this struct
 */
struct PS_UD {
  WNT_TIME AVT, StartTime;
  ULONG Warp, Share;
  struct HLS_VPROC *udvp;
  LIST_ENTRY AllThreadsEntry;
};

/*
 * utility functions to hide null pointers from the rest of the scheduler
 */

// top virtual processor to proportional share instance data
static struct PS_INSTANCE_DATA *TVPtoPSID (struct HLS_VPROC *tvp)
{
  struct PS_INSTANCE_DATA *id;
  id = (struct PS_INSTANCE_DATA *)tvp->BottomSched->SchedData;
  return id;
}

// scheduler instance pointer to PS instance data
static struct PS_INSTANCE_DATA *SItoPSID (struct HLS_SCHED_INSTANCE *si)
{
  struct PS_INSTANCE_DATA *id;
  id = (struct PS_INSTANCE_DATA *)si->SchedData;
  return id;
}

// bottom VP to PS instance data
static struct PS_INSTANCE_DATA *BVPtoPSID (struct HLS_VPROC *bvp)
{
  struct PS_INSTANCE_DATA *id;
  id = (struct PS_INSTANCE_DATA *)bvp->TopSched->SchedData;
  return id;
}

// bottom VP to per-VP data
static struct PS_UD *BVPtoPSUD (struct HLS_VPROC *bvp)
{
  struct PS_UD *ud;
  ud = (struct PS_UD *)bvp->UpperData;
  return ud;
}

// generic message pointer to PS-specific data
static struct HLS_RR_MSG *MsgToRRMsg (MsgHandle Context)
{
  struct HLS_RR_MSG *p = (struct HLS_RR_MSG *) Context;
  return p;
}

/*
```

```
 * main body of scheduler: HLS callbacks and support functions
 */

static void RegisterPS (struct PS_INSTANCE_DATA *Inst)
{
  Inst->TVP.TopSched->CB->B_RegisterVP (Inst->TVP.TopSched, Inst->Self, &Inst->TVP);

  if (strncmp (Inst->TVP.TopSched->Type, "priority", 8) == 0) {
    struct HLS_RR_MSG NewMsg;
    HLS_STATUS status;

    NewMsg.Type = RR_MSG_SETPRI;
    NewMsg.Pri = Inst->DefaultPri;
    status = Inst->TVP.TopSched->CB->B_Msg (NULL, &Inst->TVP, (MsgHandle)&NewMsg);
  }
}

static void MakeItSo (struct PS_INSTANCE_DATA *Inst,
                      WNT_TIME Now);

static HLS_STATUS PS_B_CallbackRegisterVP (struct HLS_SCHED_INSTANCE *Parent,
                                           struct HLS_SCHED_INSTANCE *Child,
                                           struct HLS_VPROC *bvp)
{
  struct PS_INSTANCE_DATA *Inst = BVPtoPSID (bvp);

  bvp->UpperData = (TDHandle) hls_malloc (sizeof (struct PS_UD));

  {
    struct PS_UD *ud = (struct PS_UD *) bvp->UpperData;
    ud->udvp = bvp;
    ud->Warp = 0;
    ud->AVT = 0;
    ud->Share = 0;
    InsertTailList(&Inst->AllThreads, &ud->AllThreadsEntry);
  }

  bvp->State = VP_Waiting;
  bvp->Proc = NO_PROC;

  if (Inst->NumThreads == 0) {
    RegisterPS (Inst);
    HLSSetTimer (-(10*HLS_MsToNT), Inst->Self);
  }
  Inst->NumThreads++;

  {
    WNT_TIME Now = HLSGetCurrentTime ();
    MakeItSo (Inst, Now);
  }

  return HLS_SUCCESS;
}

static void PS_B_CallbackUnregisterVP (struct HLS_VPROC *bvp)
```

```
{
  struct PS_INSTANCE_DATA *Inst = BVPtoPSID (bvp);
  struct PS_UD *ud = BVPtoPSUD (bvp);

  Inst->NumThreads--;
  if (Inst->NumThreads == 0) {
    Inst->TVP.TopSched->CB->B_UnregisterVP (&Inst->TVP);
  }

  RemoveEntryList (&ud->AllThreadsEntry);

  hls_free ((void *)bvp->UpperData);
}

static void GrantIt (struct PS_INSTANCE_DATA *Inst)
{
  struct HLS_VPROC *bvp;
  struct PS_UD *ud;

  /*
   * this will have already been set to the runnable bottom VP with
   * the earliest deadline
   */
  bvp = Inst->TVP.BottomData;

  ud = BVPtoPSUD (bvp);
  bvp->Proc = Inst->TVP.Proc;
  bvp->State = VP_Running;
  bvp->BottomSched->CB->T_VP_Grant (bvp);
  ud->StartTime = HLSGetCurrentTime ();
}

/*
 * update actual virtual time of a child VP
 */
static void UpdateAVT (struct PS_UD *ud,
                       WNT_TIME Now)
{
  WNT_TIME RunTime;

  RunTime = Now - ud->StartTime;
  if (RunTime < 0) {
    RunTime = 0;
  }
  ud->AVT += 1 + (RunTime / ud->Share);
}

static void PS_T_CallbackRevoke (struct HLS_VPROC *tvp)
{
  struct PS_INSTANCE_DATA *Inst = TVPtoPSID (tvp);
  WNT_TIME Now = HLSGetCurrentTime ();
  struct HLS_VPROC *Current = tvp->BottomData;

  UpdateAVT (BVPtoPSUD (Current), Now);
  Current->State = VP_Ready;
```

```
  Current->BottomSched->CB->T_VP_Revoke (Current);
  Current->Proc = NO_PROC;
  MakeItSo (Inst, Now);
}

/*
 * return effective virtual time for a child VP
 */
static WNT_TIME EVT (struct PS_UD *t)
{
  return (t->Warp > t->AVT) ? 0 : (t->AVT - t->Warp);
}

/*
 * return pointer to runnable virtual processor with smallest effective virtual time
 */
static struct HLS_VPROC *GetVPSmallestEVT (struct PS_INSTANCE_DATA *Inst)
{
  struct PS_UD *min = NULL;
  PRLIST_ENTRY ListHead = &Inst->AllThreads;
  PRLIST_ENTRY NextEntry = ListHead->Flink;

  while (NextEntry != ListHead) {
    struct PS_UD *ud = CONTAINING_RECORD (NextEntry, struct PS_UD, AllThreadsEntry);
    NextEntry = NextEntry->Flink;
    if (ud->udvp->State != VP_Waiting &&
        (!min || EVT (ud) < EVT (min))) {
      min = ud;
    }
  }

  return (min) ? min->udvp : NULL;
}

static void PS_T_CallbackGrant (struct HLS_VPROC *tvp)
{
  struct PS_INSTANCE_DATA *Inst = TVPtoPSID (tvp);

  tvp->BottomData = GetVPSmallestEVT (Inst);
  GrantIt (Inst);
}

static void MaybePreempt (struct HLS_VPROC *Current,
                          WNT_TIME Now)
{
  if (Current->State == VP_Running) {
    UpdateAVT (BVPtoPSUD (Current), Now);
    Current->State = VP_Ready;
    Current->BottomSched->CB->T_VP_Revoke (Current);
    Current->Proc = NO_PROC;
  }
}

/*
 * generic reschedule
```

```
 */
static void MakeItSo (struct PS_INSTANCE_DATA *Inst,
                      WNT_TIME Now)
{
  struct HLS_VPROC *bvp = Inst->TVP.BottomData;
  struct HLS_VPROC *Next;
  struct HLS_VPROC *Current;

  if (bvp) {
    if (bvp->State == VP_Running) {
      struct PS_UD *ud = BVPtoPSUD (bvp);
      UpdateAVT (ud, Now);
      ud->StartTime = Now;
    }
  }

  if (Inst->TVP.State != VP_Ready) {
    Next = GetVPSmallestEVT (Inst);
  } else {
    Next = NULL;
  }
  Current = Inst->TVP.BottomData;

  if (Current) {
    if (Next) {
      if (Current != Next) {
        MaybePreempt (Current, Now);
        Inst->TVP.BottomData = Next;
        GrantIt (Inst);
      }
    } else {
      MaybePreempt (Current, Now);
      if (Inst->TVP.State != VP_Ready) {
        Inst->TVP.TopSched->CB->B_VP_Release (&Inst->TVP);
      }
      Inst->TVP.BottomData = NULL;
    }
  } else {
    if (Next) {
      Inst->TVP.TopSched->CB->B_VP_Request (&Inst->TVP);
    } else {
      Inst->TVP.BottomData = NULL;
      if (Inst->TVP.State == VP_Ready && !GetVPSmallestEVT(Inst)) {
        Inst->TVP.TopSched->CB->B_VP_Release (&Inst->TVP);
      }
    }
  }
}

/*
 * compute system virtual time - only used when a VP unblocks
 */
static WNT_TIME SVT (struct PS_INSTANCE_DATA *Inst)
{
  struct PS_UD *min = NULL;
```

```
  PRLIST_ENTRY ListHead = &Inst->AllThreads;
  PRLIST_ENTRY NextEntry = ListHead->Flink;

  while (NextEntry != ListHead) {
    struct PS_UD *ud = CONTAINING_RECORD (NextEntry, struct PS_UD, AllThreadsEntry);
    NextEntry = NextEntry->Flink;
    if (ud->udvp->State != VP_Waiting &&
        (!min || ud->AVT < min->AVT)) {
      min = ud;
    }
  }
  return (min) ? min->AVT : 0;
}

static void PS_B_CallbackRequest (struct HLS_VPROC *bvp)
{
  struct PS_UD *ud = BVPtoPSUD (bvp);
  struct PS_INSTANCE_DATA *Inst = BVPtoPSID (bvp);
  WNT_TIME Now = HLSGetCurrentTime ();
  WNT_TIME tSVT;

  tSVT = SVT (Inst);
  ud->AVT = max (ud->AVT, tSVT);
  bvp->State = VP_Ready;
  MakeItSo (Inst, Now);
}

static void PS_B_CallbackRelease (struct HLS_VPROC *bvp)
{
  struct PS_INSTANCE_DATA *Inst = BVPtoPSID (bvp);
  WNT_TIME Now = HLSGetCurrentTime ();

  if (bvp->State == VP_Running) {
    UpdateAVT (BVPtoPSUD(bvp), Now);
  }

  bvp->State = VP_Waiting;
  bvp->Proc = NO_PROC;
  MakeItSo (Inst, Now);
}

static void PS_I_TimerCallback (struct HLS_SCHED_INSTANCE *Self)
{
  struct PS_INSTANCE_DATA *Inst = SItoPSID (Self);
  WNT_TIME Now = HLSGetCurrentTime ();

  if (Inst->NumThreads == 0) {
    return;
  }

  MakeItSo (Inst, Now);
  HLSSetTimer (-(10*HLS_MsToNT), Inst->Self);
}

static HLS_STATUS PS_B_CallbackMsg (struct HLS_SCHED_INSTANCE *InstArg,
```

```
                                    struct HLS_VPROC *bvp,
                                    MsgHandle Msg)
{
  struct PS_INSTANCE_DATA *Inst;
  struct HLS_RR_MSG *m;
  HLS_STATUS status;

  if (bvp) {
    Inst = BVPtoPSID (bvp);
  } else {
    Inst = SItoPSID (InstArg);
  }

  m = MsgToRRMsg (Msg);

  switch (m->Type) {
  case RR_MSG_SETDEFPRI:
    Inst->DefaultPri = m->Pri;
    status = HLS_SUCCESS;
    break;
  case RR_MSG_SETPRI:
    {
      struct PS_UD *ud = BVPtoPSUD (bvp);

      ud->Share = m->Pri * 10;
      if (ud->Share == 0) {
        ud->Share = 10;
      }
      status = HLS_SUCCESS;
    }
    break;
  case RR_MSG_SETRR:
    status = HLS_SUCCESS;
    break;
  case RR_MSG_SETSHARE:
    {
      struct PS_UD *ud = BVPtoPSUD (bvp);

      ud->Warp = m->Warp;
      ud->Share = m->Share;
      status = HLS_SUCCESS;
    }
    break;
  case RR_MSG_NULL:
    status = HLS_SUCCESS;
    break;
  default:
    status = HLS_INVALID_PARAMETER;
  }

  return status;
}

static void PS_I_Init (struct HLS_SCHED_INSTANCE *Self,
                       struct HLS_SCHED_INSTANCE *Parent)
```

```
{
  Self->SchedData = (SDHandle) hls_malloc (sizeof (struct PS_INSTANCE_DATA));
  {
    struct PS_INSTANCE_DATA *Inst = (struct PS_INSTANCE_DATA *)Self->SchedData;

    InitializeListHead (&Inst->AllThreads);
    Inst->PSInitState = 1;
    Inst->NumThreads = 0;
    Inst->TimeIncrement = KeQueryTimeIncrement ();

    Inst->TVP.State = VP_Waiting;
    Inst->TVP.BottomData = NULL;
    Inst->TVP.BottomSched = Self;
    Inst->TVP.TopSched = Parent;

    strcpy (Self->Type, "priority");
    Inst->DefaultPri = -1;
    Inst->Self = Self;
  }
}

static void PS_I_Deinit (struct HLS_SCHED_INSTANCE *Self)
{
  hls_free ((void *)Self->SchedData);
}

struct HLS_CALLBACKS PS_CB = {
  "PS",
  PS_B_CallbackRegisterVP,
  PS_B_CallbackUnregisterVP,
  PS_B_CallbackRequest,
  PS_B_CallbackRelease,
  PS_B_CallbackMsg,
  PS_T_CallbackGrant,
  PS_T_CallbackRevoke,
  PS_I_Init,
  PS_I_Deinit,
  PS_I_TimerCallback,
};
```

Bibliography

[1] Tarek F. Abdelzaher, Ella M. Atkins, and Kang Shin. QoS Negotiation in Real-Time Systems and its Application to Automated Flight Control. In *Proc. of the 3rd IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[2] Luca Abeni and Giorgio Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, December 1998.

[3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[4] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[5] Gaurav Banga, Peter Druschel, and Jeffery C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[6] Luciano P. Barreto and Gilles Muller. Bossa: a DSL framework for application-specific scheduling policies. Research Report 1384, INRIA, Rennes, France, April 2001.

[7] Craig Barrett. Keynote Speech at COMDEX, January 2001. `http://www.intel.com/pressroom/archive/speeches/crb20010105ces.htm`.

[8] Andy C. Bavier, A. Brady Montz, and Larry L. Peterson. Predicting MPEG Execution Times. In *Proc. of the Joint International Conf. on Measurement and Modeling of Computer Systems*, pages 131–140, Madison, WI, June 1998.

[9] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[10] Andrew Bettison, Andrew Gollan, Chris Maltby, and Neil Russell. Share II – A User Administration and Resource Control System for UNIX. In *Proc. of the 5th Systems Administration Conf. (LISA V)*, September 1991.

[11] Gregory Bollella and Kevin Jeffay. Support For Real-Time Computing Within General Purpose Operating Systems: Supporting Co-Resident Operating Systems. In *Proc. of the 1st IEEE Real-Time Technology and Applications Symposium*, pages 4–14, Chicago, IL, May 1995.

[12] Scott Brandt, Gary Nutt, Toby Berk, and Marty Humphrey. Soft Real-Time Application Execution with Dynamic Quality of Service Assurance. In *Proc. of the 6th International Workshop on Quality of Service*, pages 154–163, Napa, CA, 1998.

[13] Scott Brandt, Gary Nutt, Toby Berk, and James Mankovich. A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 307–317, Madrid, Spain, December 1998.

[14] George M. Candea and Michael B. Jones. Vassal: Loadable Scheduler Support for Multi-Policy Scheduling. In *Proc. of the 2nd USENIX Windows NT Symposium*, pages 157–166, Seattle, WA, August 1998.

[15] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, pages 45–58, San Diego, CA, October 2000.

[16] Hao-hua Chu and Klara Nahrstedt. CPU Service Classes for Multimedia Applications. In *Proc. of the 6th IEEE International Conf. on Multimedia Computing and Systems*, pages 2–11, Florence, Italy, June 1999.

[17] Erik Cota-Robles and James P. Held. A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[18] Zhong Deng, Jane W.-S. Liu, Lynn Zhang, Seri Mouna, and Alban Frei. An Open Environment for Real-Time Applications. *Real-Time Systems Journal*, 16(2/3):165–185, May 1999.

[19] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 261–276, Seattle, WA, October 1996.

[20] Kenneth J. Duda and David C. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[21] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[22] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for Kernel and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, Saint-Malô, France, October 1997.

[23] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proc. of the USENIX Winter 1994 Technical Conf.*, San Francisco, CA, January 1994.

[24] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, October 1996.

[25] Borko Furht. Processor Architectures for Multimedia. In *Handbook of Multimedia Computing*, pages 447–467. CRC Press, 1999.

[26] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4, An Open Systems Design*. Prentice-Hall, 1994.

[27] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 107–121, Seattle, WA, October 1996.

[28] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-Time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 157–168, Stanford, CA, August 1996.

[29] Hewlett-Packard Company Information Networks Division. Netperf: A Network Performance Benchmark, 1995. `http://www.netperf.org/netperf/training/Netperf.html`.

[30] David Hull, Wu-chun Feng, and Jane W.-S. Liu. Operating System Support for Imprecise Computation. In *Proc. of the AAAI Fall Symposium on Flexible Computation*, pages 9–11, Cambridge, MA, November 1996.

[31] David Ingram. Soft Real Time Scheduling for General Purpose Client-Server Systems. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, March 1999.

[32] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

[33] Kevin Jeffay and Steve Goddard. A Theory of Rate-Based Execution. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 304–314, Phoenix, AZ, December 1999.

[34] Kevin Jeffay, F. Donelson Smith, Arun Moorthy, and James Anderson. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 480–491, Madrid, Spain, December 1998.

[35] Kevin Jeffay and Donald L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In *Proc. of the Real-Time Systems Symposium*, pages 212–221, Raleigh-Durham, NC, December 1993.

[36] Michael B. Jones, Paul J. Leach, Richard P. Draves, and Joseph S. Barrera, III. Modular Real-Time Resource Management in the Rialto Operating System. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[37] Michael B. Jones and John Regehr. CPU Reservations and Time Constraints: Implementation Experience on Windows NT. In *Proc. of the 3rd USENIX Windows NT Symposium*, pages 93–102, Seattle, WA, July 1999.

[38] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, pages 96–101, Rio Rico, AZ, March 1999.

[39] Michael B. Jones, John Regehr, and Stefan Saroiu. Two Case Studies in Predictable Application Scheduling Using Rialto/NT. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.

[40] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, Saint-Malô, France, October 1997.

[41] Michael B. Jones and Stefan Saroiu. Predictability Requirements of a Soft Modem. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Cambridge, MA, June 2001.

[42] Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

[43] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, June 1997.

[44] Eddie Kohler, Benjie Chen, M. Frans Kaashoek, Robert Morris, and Massimiliano Poletto. Programming language techniques for modular router configurations. Technical Report MIT-LCS-TR-812, MIT Laboratory for Computer Science, August 2000.

[45] Tei-Wei Kuo and Ching-Hui Li. A Fixed-Priority-Driven Open Environment for Real-Time Applications. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 256–267, Phoenix, AZ, December 1999.

[46] Intel Architecture Labs. Tools and Techniques for Softmodem IHV/ISV Self-Validation of Compliance with PC 99 Guidelines for Driver-Based Modems. Technical report, Intel Corporation, July 1998.

[47] Kam Lee. Performance Bounds in Communication Networks with Variable-Rate Links. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 126–136, Cambridge, MA, August 1995.

[48] John Lehoczky, Lui Sha, and Ye Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proc. of the Real-Time Systems Symposium*, pages 166–171, Santa Monica, CA, December 1989.

[49] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[50] Chih-han Lin, Hao-hua Chu, and Klara Nahrstedt. A Soft Real-time Scheduling Server on the Windows NT. In *Proc. of the 2nd USENIX Windows NT Symposium*, Seattle, WA, August 1998.

[51] Giuseppe Lipari and Sanjoy K. Baruah. Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems. In *Proc. of the 6th IEEE Real-Time Technology and Applications Symposium*, pages 166–175, Washington DC, May 2000.

[52] Giuseppe Lipari, John Carpenter, and Sanjoy K. Baruah. A Framework for Achieving Inter-Application Isolation in Multiprogrammed Hard Real-Time Environments. In *Proc. of the 21st IEEE Real-Time Systems Symposium*, pages 217–226, Orlando FL, November 2000.

[53] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[54] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[55] C. Douglass Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986. Also appears as Technical Report CMU-CS-86-134.

[56] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. The Design and Evaluation of a Feedback Control EDF Scheduling Algorithm. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 56–67, Phoenix, AZ, December 1999.

[57] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[58] Richard McDougall, Adrian Cockcroft, Evert Hoogendoorn, Enrique Vargas, and Tom Bialaski. *Resource Management*. Sun Microsystems Press, 1999.

[59] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proc. of the IEEE International Conf. on Multimedia Computing and Systems*, May 1994.

[60] Cyril Meurillon. Be Engineering Insights: The Kernel Scheduler and Real-Time Threads. *Be Newsletter*, 37, August 1996. `http://www-classic.be.com/aboutbe/benewsletter/Issue37.html`.

[61] Microsoft. Windows 2000 DDK Documentation, MSDN Library, March 2000. `http://support.microsoft.com/support/kb/articles/q186/7/75.asp`.

[62] Jeff Mogul and Anita Borg. The Effect of Context Switches on Cache Performance. In *Proc. of the 4th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA, April 1991.

[63] Aloysius Mok and Deji Chen. A Multiframe Model for Real-Time Tasks. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

[64] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 153–167, Seattle, WA, October 1996.

[65] Klara Nahrstedt and Jonathan Smith. The QoS Broker. *IEEE MultiMedia*, 2(1):53–67, Spring 1995.

[66] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proc. of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[67] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malô, France, October 1997.

[68] Shuichi Oikawa and Ragunathan Rajkumar. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proc. of the 5th IEEE Real-Time Technology and Applications Symposium*, pages 111–120, Vancouver, BC, Canada, June 1999.

[69] David Olofson. Personal communication, April 2000.

[70] Don Oparah. A Framework for Adaptive Resource Management in a Multimedia Operating System. In *Proc. of the 6th IEEE International Conf. on Multimedia Computing and Systems*, Florence, Italy, June 1999.

[71] David Petrou, John W. Milford, and Garth A. Gibson. Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. In *Proc. of the USENIX 1999 Annual Technical Conf.*, pages 1–14, Monterey, CA, June 1999.

[72] Krithi Ramamritham, Chia Shen, Oscar Gonzalez, Shubo Sen, and Shreedhar B. Shirgurkar. Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations. In *Proc. of the 4th IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998.

[73] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A Foundation for Open Systems. In *Proc. of the 3rd Workshop on Workstation Operating Systems*, September 1989.

[74] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component Composition for Systems Software. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, October 2000.

[75] Daniela Roşu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. FARA - A Framework for Adaptive Resource Allocation in Complex Real-Time Systems. In *Proc. of the 4th IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998.

[76] Lui Sha, Raj Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[77] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.

[78] Steven Sommer. Removing Priority Inversion from an Operating System. In *Proc. of the 19th Australasian Computer Science Conf.*, pages 131–139, Melbourne, Australia, January 1996.

[79] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems Journal*, 10(1):179–210, 1996.

[80] Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari, and Douglas Niehaus. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. In *Proc. of the 4th IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998.

[81] John A. Stankovic and Krithi Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, May 1991.

[82] John A. Stankovic, Krithi Ramamritham, Douglas Niehaus, Marty Humphrey, and Gary Wallace. The Spring System: Integrated Support for Complex Real-Time Systems. *Real-Time Systems Journal*, 16(2/3):223–251, May 1999.

[83] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[84] Ian Stoica, Hussein Abdel-Wahab, and Kevin Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proc. of Multimedia Computing and Networking 1997*, pages 207–214, San Jose, CA, February 1997.

[85] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, Washington, DC, December 1996.

[86] TimeSys. The Concise Handbook of Linux for Embedded Real-Time Systems, 2000. `ftp://ftp.timesys.com/pub/docs/LinuxRTHandbook.pdf`.

[87] Ken W. Tindell, Alan Burns, and Andy J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems Journal*, 6(2):133–152, March 1994.

[88] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proc. of the 1st USENIX Mach Workshop*, pages 73–82, October 1990.

[89] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, February 1995.

[90] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, pages 1–11. USENIX Association, 1994.

[91] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1994.

[92] Yu-Chung Wang and Kwei-Jay Lin. Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 246–255, Phoenix, AZ, December 1999.

[93] Victor Yodaiken. The RTLinux Manifesto. In *Proc. of The 5th Linux Expo*, Raleigh, NC, March 1999.