A Unified Scheduling Model for Precise Computation Control

by

Michael Frisbie

B.S. (Computer Science), University of Kansas, Lawrence, Kansas, 2002

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

To Steve and Carolyn Frisbie

Acknowledgments

I would like to thank Dr. Douglas Niehaus, my advisor and committee chair, for providing guidance during the work presented here. I would also like to thank Dr. David Andrews and Dr. Jerry James for serving as members of my thesis committee.

I especially thank Jacob Woltersdorf for his work on the original implementation of this project. I thank Hariharan Subramanian, Deepti Mokkapati, and Tejasvi Aswathanarayana for their assistance and for keeping me smiling. I also want to acknowledge Mitchell Trope and Thomas Gleixner for their help and Will Dinkel for his early guidance in the KURT-Linux project.

My appreciation for the work of Linus Torvalds and his fellow contributors to the Linux kernel cannot be adequately expressed. I am also grateful to the GNU Project for producing much of the free software that I use.

This document was edited with XEmacs and typeset with $\LaTeX$. All figures were drawn using Xfig, and all graphs were generated by gnuplot. Quantitative data was gathered and processed using the Data Streams tool set.

Abstract

Computer system designers wish to directly specify the computational semantics they desire with as much precision as possible. Because current operating systems make assumptions about the behavioral constraints of their applications, this is often unattainable. Each set of semantics imposes a set of constraints, and satisfaction of any given constraint may require precise control over any arbitrary set of system resources. Therefore, a configurable solution that allows control of all system computations is required. The unified scheduling model makes this possible by providing a configurable hierarchy that explicitly states the relationship between scheduling algorithms and the computations that they control.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Computer system designers wish to directly specify the computational semantics they desire with as much precision as possible. Because current operating systems make assumptions about the behavioral constraints of their applications, this is often unattainable. System designers are forced to map their desired semantics onto the interface that is provided by the operating system. If the interface does not provide the precision required by the desired semantics, then the system designer is forced to compromise.

Traditionally, computer systems have been divided into two separate use categories: general-purpose and real-time. General-purpose operating systems, by definition, are not specialized for any particular application. Therefore, they make assumptions that optimize for average-case execution. Real-time and embedded systems, on the other hand, are specialized for specific applications and impose constraints that account for worst-case execution behavior. Because of the clear partition between these categories, operating systems are usually designed to service only one of them.

Because computer systems are applied in an increasingly wide range of environments, a single set of system characteristics is no longer appropriate for every application. System designers are thus increasingly interested in configurable platforms that enable them to directly choose the set of characteristics that most closely match their requirements.

Precise control over all system resources is an important goal of systems that must satisfy specific constraints. The reason for this is that satisfaction of any given constraint may require precise control over any arbitrary set of system resources. Therefore, resource scheduling methods are among the strongest influences on system behavior, and scheduling of the central processing unit (CPU) is the most obvious resource-control function in any computer system. As such, it has been the subject of an enormous amount of theoretical research and practical implementation.

Most approaches to scheduling concentrate on how application threads use the CPU, but threads are not the only computational elements managed by most operating systems. The most obvious example of non-thread computational elements managed by the operating system are interrupt service routines, which are generally scheduled under an "as fast as possible" policy. Most scheduling approaches are well-developed in their management of thread execution, but methods for explicitly controlling other types of computational elements are far less common and less well-developed.

Explicitly representing and fully integrating the execution scheduling of all computational elements on the system is an important issue because it is a necessary step in increasing the precision with which use of the CPU resource can be controlled. All computation components on the system affect system behavior and should be accessible under the system's scheduling model. For example, in systems where interrupt processing is not integrated with application thread scheduling, interrupt processing typically manifests as jitter in the thread scheduling accuracy. Increasing resource scheduling precision will thus require creating a scheduling policy framework that fully integrates the control of all computational elements. There are several types of computational elements in a typical operating system. For example, the computational elements present in the Linux operating system include immediate interrupt handlers (hardirqs), deferred interrupt handlers (softirqs), tasklets, and processes [3][2].

Scheduling policies can be classified according to how explicitly they represent and expose to the user the policies controlling execution of the system's computational elements. Scheduling policies in most computer systems limit user access to only the thread scheduling policy, keeping the policies controlling other computational elements hidden within the operating system. Further, the rules governing the interaction among the policies controlling the scheduling of different computational element types are often left implicitly defined. In standard Linux, policies controlling CPU usage by each of the computational element categories are implemented by independent parts of the source code. This makes the interaction of these policies difficult to both analyze and alter. Further, their interaction is often emergent behavior which results from operating system code implementation and structure rather than an explicitly specified control algorithm. Through code evolution, the policies controlling these computation components can change, not only as a decision to modify policy, but also as a side-effect of other, possibly unrelated, decisions.

Current practice does include some methods by which developers can influence the scheduling policies controlling interrupt handling. For example, developer configuration of hardware interrupt levels and handler interrupt level assignments is an indirect and limited form of influence over operating system interrupt processing policy. Some current operating systems attempt to increase scheduling configurability by bringing interrupt handler execution under their thread scheduling model [10][19]. Such systems typically assign execution of one or more interrupt service routines to specific threads with individual priorities. This allows the user to access the interrupt handler scheduling model using the available mechanisms for thread scheduling. However, this technique is not adequate for all desirable system semantics, as it does not allow for multiple scheduling policies or access to the scheduling model of all computation types.

A fully configurable system should implement the scheduling policies controlling every computational element of the system as an explicit scheduling decision function (SDF) and should expose the semantics of each SDF as a configurable system property. Further, the best approach is to combine the scheduling policies controlling execution of all types of computational elements into a unified and fully-configurable scheduling framework. Each SDF is associated with a set of computations, and this association is represented in a structure known as a group. Application orientation is a key feature of this unified model, because the group structure reflects application computation structure and the associated scheduling decision functions permit application-specific scheduling support.

Computation components are controlled under a scheduling hierarchy. The hierarchy embodies the wishes of the implementer about the policies controlling resource use by computation components. The normal caveat about the potentially large difference between what the developer thought he said and what he actually said applies. This model provides a substantial advantage to the developer in this regard, compared to conventional systems, in that the semantics of the group scheduling hierarchy can more closely correspond to the semantics the developer desires.

One necessary prerequisite of a unified scheduling model is that Linux interrupt processing be restructured in order to place it under configurable scheduling control. The current approach to interrupt handler execution control, which uses hardware enabling and disabling mechanisms, limits the responsiveness of Linux as a real-time operating system and limits the system's ability to control interrupt processing under configurable semantics. By moving interrupt control from hardware to software, the flexibility that software provides can be harnessed.

The rest of the thesis first discusses related work in Chapter 2 and then describes the implementation of the unified scheduling model in Chapter 3. Chapter 4 presents the experimental results demonstrating the validity of the claims that have been made about this model, while Chapter 5 presents conclusions and discusses future work.

Chapter 2

Related Work

The operating system is responsible for executing tasks that the user specifies. This is most clearly accomplished by executing processes, or threads. These processes run programs that exist on the user's system. Because users typically wish to execute more than one program at a time, the operating system must decide, or schedule, how to share the available resources among the available threads.

2.1 Thread Scheduling

The problem of scheduling all available system resources can be initially simplified by considering only CPU scheduling. All resources on the system are used by a set of computation components, and resource semantics impose constraints on how computations use them. For example, some resources are preemptible, which means that control of them can be involuntarily taken away, while others are not. If these constraints are incorporated into the CPU scheduling model, then there is no need to implement separate schedulers for each resource.

Therefore, most scheduling research is focused on allocating use of the CPU. Although this limited scope is understandable, since it greatly simplifies the problem, complications such as priority inversion, which is discussed in Section 2.1.2, have arisen as a result.

Furthermore, CPU scheduling research has generally been restricted to the scheduling of threads. This is because thread scheduling has the most dramatic effect on overall system performance. As a result, a myriad of algorithms for this specific purpose have been developed.

2.1.1 Priority Scheduling

Almost all modern operating systems utilize a priority scheduling algorithm in their thread scheduler. There are many variations of this algorithm, but all implementations require that each thread be given a particular integer priority. When a thread is created, it is either given a predetermined, default priority, or it inherits the priority of its parent thread. This priority can then be modified by the user in order to indicate the importance of the task, and this is the extent to which users are given access to the thread scheduling algorithm.

The scheduler considers the priority of each task, and the task with the highest priority is selected for execution. Many general-purpose operating systems also implement a technique known as priority aging [16]. Aging allows low-priority processes to make adequate progress. After each scheduling decision, each process that is not selected has its priority increased by some amount, thereby increasing its probability of being selected in the next scheduling decision. A process's priority is returned to its original value once it is selected. Eventually, under most schemes, every process is guaranteed to acquire the highest priority and be chosen to execute.

Priority-driven preemptive scheduling utilizes the priority scheduling algorithm. In addition, when a process with higher priority than the currently running process becomes available for execution, the currently running process is preempted. Priority-driven preemptive scheduling is popular because other scheduling algorithms can be mapped onto it.

The rate-monotonic algorithm is one example of a scheduling algorithm that can be mapped onto priority-driven preemptive scheduling. This is done by assigning higher priorities to processes with shorter periods [11]. The rate-monotonic mapping is a static priority assignment, since a process's priority only needs to be assigned once.

The earliest-deadline-first algorithm is another example of a scheduling algorithm that can be mapped onto priority-driven preemptive scheduling [9]. This is accomplished by assigning higher priorities to tasks whose deadlines are nearer [11]. This mapping is a dynamic priority assignment, since the priority of each task must be reassigned as time passes. Dynamic priority assignment is undesirable, as it requires that new priorities be assigned as time passes. As a result, additional overhead is incurred. This drawback is one reason why although, in theory, it might be possible to map a particular algorithm onto priority scheduling, it may also be unsatisfactory to do so in practice.

There are many reasons why priority scheduling has become a standard in operating system schedulers. First, any given thread scheduling semantics can be represented by specifying a particular priority arrangement. However, the ease with which any given semantics can be translated into a priority arrangement can vary greatly. Second, priority schedulers are efficient, both in terms of speed and space. The set of information that needs to be considered in order to make a decision is limited to a set of integers, and finding the highest integer in the set can be accomplished quickly. Finally, priority schedulers are easy to implement due to their limited scope and straightforward semantics.

2.1.2 Priority Inversion

Priority inversion occurs when a high priority thread cannot execute because it requires a resource that is held by a low priority thread. Because the high priority thread must wait for the low priority thread to finish using the resource, the high priority thread during this time period does not benefit from having a high priority.

One solution to this problem is known as priority inheritance. When a high priority thread must wait on a particular resource that is held by low priority thread, the priority of the thread holding the resource is temporarily increased to that of the high priority thread. When the resource is released by the thread, its priority is returned to its original value.

Although there are fixes for priority inversion, the fundamental problem that it presents is still left unaddressed. The thread scheduler's view of the system is simply a set of priorities. Anything outside of this view, such as resource usage, is ignored. The unified scheduling model, because of its configurable nature, allows for an arbitrarily large, or small, amount of data to be considered in the scheduling decision. Problems such as priority inversion can be avoided by explicitly checking for them, rather than creating indirect methods that correct them.

2.2 Co-Resident Operating Systems

Co-resident operating systems are one popular approach to unifying the guarantees provided by a real-time system with the capabilities of a general-purpose operating system. This architecture allows for a real-time executive to coexist with a general-purpose operating system. Resources are then divided in such a way that the real-time kernel can meet its timing constraints, and the general-purpose kernel can operate with little modification [1].

This approach is implemented in the RTLinux system [20]. RTLinux places a small, real-time operating system underneath the Linux kernel. This small executive treats the Linux kernel as the “idle” task, meaning that it is executed whenever the executive does not have any real-time tasks to execute. RTLinux accomplishes this by intercepting interrupt signals. When an interrupt is asserted, the RTLinux executive determines if it is related to any real-time tasks. If so, then they are executed. Otherwise, handling of the interrupt is passed on to the Linux kernel. Because of this, Linux interrupt control operations, such as *cli* and *sti*, no longer operate directly on the hardware, as this would interfere with the RTLinux executive. Instead, these control operations communicate to the executive whether interrupt signals should be delivered immediately, as is the case when the Linux kernel has enabled interrupts, or whether they should be delayed, as is the case when the Linux kernel has disabled interrupts.

RTLinux is not adequate for expressing the design semantics of many applications and is inconvenient for others. This is a result of the fact that the system makes assumptions about what constraints will need to be enforced. Computations are assumed to either have strict timing constraints that cannot be violated or no timing constraints at all. Although, this programming model fits many applications, it is not sufficient in all cases. Further, real-time tasks are not executed by the Linux kernel and must therefore be specifically written to run under the RTLinux executive. As a result, these tasks cannot directly take advantage of the features provided by the Linux kernel.

Because the co-resident operating systems approach requires that each operating system be partitioned from every other operating system, interaction among the resident kernels is restricted. The ability to communicate information between kernels is severely limited, and the capabilities provided by each operating system are difficult to use across partition boundaries.

2.3 POSIX Real-Time

The POSIX standard provides facilities for soft real-time capability [4]. Among other features, it specifies special real-time priority classes, named SCHED_FIFO, SCHED_RR, and SCHED_OTHER, that allow tasks to be guaranteed specific treatment by the operating system's scheduler.

The POSIX real-time scheduling capabilities suffer from the same problems as other priority scheduling policies, which are discussed in Section 2.1.1. However, the POSIX real-time scheduling is interesting in that it creates a scheduling hierarchy among the FIFO, round-robin, and system scheduling policies. Although this hierarchy is not configurable and controls only thread scheduling, it demonstrates how a single scheduling policy is not sufficient for all applications.

2.4 High-Resolution Timers

The UTIME project increases the temporal resolution of Linux [17]. As a result, the precision with which Linux timer hardirqs, and hence time-triggered computations, can be executed is enhanced. However, the accuracy of these time-triggered computations is adversely affected by the lack of control over various other aspects of the system.

The UTIME project allows for timer interrupts to be scheduled with high precision. Since Linux is a general-purpose operating system, it places the system's timer chip in periodic mode. This mode triggers a timer interrupt periodically at a predetermined rate. Although this rate could simply be raised in order to increase temporal resolution, the overhead associated with processing timer interrupts would become unacceptable at some point. Therefore, the UTIME project switches the timer chip from periodic to one-shot mode [6]. One-shot mode allows for the timer chip to be programmed to interrupt the CPU at a particular time. Thus, instead of interrupting the CPU when there is a possibility that a timer computation could be executed, it instead interrupts the CPU whenever a timer computation actually needs to be executed. The downside to this alteration is that the timer chip must be reprogrammed for each interrupt.

Although UTIME increases the resolution with which timer interrupts can be scheduled, the actual execution of the timer hardirq and timer computations can be affected by many other factors. First, execution of the timer hardirq can be prevented by interrupt concurrency control mechanisms, which are discussed in Section 3.1.2. Second, Linux timer computations are executed in the timer bottom-half. Since bottom-halves can potentially be deferred for long periods of time, the accuracy with which timer computations are executed can be significantly decreased.

2.5 Real-Time Operating Systems

In its broadest definition, real-time systems are computer systems whose behavior is dependent not only on logical correctness of computational results but also on the time at which they are produced [8]. As the behavior of programs that general-purpose systems execute are becoming increasingly dependent on correct timing, such as multimedia applications, this definition is becoming less useful for classifying an entire computer system. Thus, applications can be categorized based on the types of timing guarantees that they require, and real-time operating systems can be categorized based on the types of applications for which they can provide the necessary guarantees. Two such categories of applications are commonly used:

- Hard real-time applications must meet their timing constraints requirements. Failure to meet a timing constraint can result in a critical, unrecoverable error.

The result of a hard real-time computation that does not meet its timing constraints is therefore worthless.

- Soft real-time applications can afford violations of their timing constraints. If a soft real-time application fails to meet its timing constraints, then there may be undesirable effects, but the failure does not have critical, unrecoverable consequences. Thus, the result of a soft real-time computation may still possess value even if its timing constraints are violated.

Numerous techniques have been adopted by real-time operating system developers in order to meet the demands created by these two types of applications. LynxOS [10] and Timesys Linux [19] both encapsulate interrupt service routines as kernel threads*. This eliminates much of the unpredictability that is introduced by interrupts. Processes share a single address space with the kernel in the VxWorks [18] operating system. This gives tasks direct access to system resources. RTAI [15] provides a hardware abstraction layer within the Linux kernel. This layer replaces some of the parts of Linux whose functionality must be modified in order to provide real-time constraints.

2.6 Hierarchical Scheduling

Hierarchical process scheduling is not a new concept. The Hierarchical Loadable Schedulers (HLS) effort allows scheduling algorithms and threads to be composed in a similar hierarchical fashion [14][12]. Each scheduler within the HLS hierarchy provides guarantees to its children, and these guarantees cannot be stronger than the guarantee given to its parent schedulers. The strongest guarantee is given to the root scheduler, which is guaranteed 100% of the CPU. The HLS process hierarchy, while similar in structure the unified scheduling model presented here, does not incorporate other types of computation such as interrupt service routines, which might affect the guarantees that can be provided to each application.

Further development of HLS has been done to incorporate various types of system

*A kernel thread is a process that executes entirely within the address space of the kernel.

computations, such as hardirqs, softirqs, and processes into a single scheduling hierarchy [13]. This work focuses on the ability of a hierarchical, unified scheduling model to facilitate policy changes in real-time systems. However, it is focused simply on the ability to easily move a computation from one execution environment to another. For example, HLS provides the ability to easily change a bottom-half computation into a process in order to modify the semantics of its execution. The unified scheduling model presented here focuses on the ability of the system designer to explicitly state constraints without necessarily adhering to a predefined set of assumptions.

Chapter 3

Implementation

The unified scheduling model consists of a set of components and permits the system designer to configure them into a hierarchic decision structure that is responsible for deciding which computation should be executed on a particular CPU. Each computation is represented in this structure, and specific sections of the hierarchy are responsible for deciding when to execute it. This is accomplished by associating a scheduling decision function (SDF) with a particular set of computations. This combination is represented in a structure known as a group, and these groups are linked together to form a hierarchy. Many computation types have been adapted for inclusion in this hierarchy. As a result, a unified and configurable model of scheduling has been established for Linux.

A particularly important aspect of this approach is that it places hardirqs under control of this unified scheduling model. The simple mechanism of disabling the occurrence of interrupts at the hardware level no longer provides the flexibility that is needed in order to create a scheduling decision function for hardirqs. The flexibility created by this transformation allows for the development of scheduling algorithms for hardirq control beyond the traditional "as fast as possible" scheme.

3.1 Existing Scheduling and Execution Mechanisms

There are many types of computation in the Linux kernel. Each of these computation types has a unique method for scheduling and execution. One aspect of all computations that significantly affects how they are scheduled and executed is their context. A context usually consists of an address space, kernel stack, user stack*, and register set. Computations such as processes possess their own context. A set of threads in the familiar multi-threaded programming model each have their own stacks while sharing the same address space. Other computations, such as interrupt service routines, simply execute in whichever context the system is currently running when the interrupt occurs. While CPU usage accounting for computations that possess their own context is a standard feature in most operating systems and is typically necessary information for the process scheduler, much less sophisticated accounting, if any, exists for context-borrowing computations. Essentially, they are off-the-books computations which are simply assumed to be scheduled and executed correctly without any acknowledged effect on the system.

3.1.1 Processes

Processes are scheduled using a dynamic priority policy. Each process is assigned a priority value, which is specified by the user, that is used to indicate its importance. This value, along with several other factors, is taken into consideration whenever process scheduling takes place.

Process scheduling can be initiated either actively or lazily. Active initiation is accomplished by directly invoking the scheduler function. Lazy initiation is accomplished by first setting the *need_resched* flag. There are certain places in the the kernel where this flag is checked. If the flag is set, then the process scheduler is called.

In older versions of the Linux kernel, this flag was only checked when returning from supervisor mode to user mode. However, with the introduction of the preemption patch, this flag is checked at various control points within the kernel. This allows

*Kernel threads, which execute entirely within the kernel address space, do not require a user stack.

for decreased scheduling latency, which is described in Section 3.2.2.

The process scheduler selects the process with the highest “goodness” value for execution. The calculation of this value is constantly evolving, but it typically involves a thread’s specified priority, its state, its CPU affinity, and the length of time since it was last executed.

After selecting a process, the system stores the current execution context and switches to the context of the selected process. This involves, among other things, restoring the saved register values of the process and its memory map. Execution then continues in the context of the selected process.

3.1.2 Hardirqs

Hardirqs are handled using an “as fast as possible” strategy. Many components, both in hardware and software, interact to control the execution of hardirqs. The lowest level of control is resident in the Programmable Interrupt Controller (PIC). This device is responsible for queuing interrupt signals and delivering them to the CPU. The x86 PIC uses a static priority policy when determining which interrupt signal should be given to the CPU. If an interrupt with higher priority than the currently executing interrupt is asserted, then it is immediately sent to the CPU. Otherwise, it is queued for later delivery. Once an interrupt has been handled by the CPU, it is responsible for sending an end-of-interrupt (EOI) signal to the PIC. The PIC then sends the pending interrupt signal with highest priority to the CPU.

Beyond sending an EOI signal, the CPU has further control over interrupts. It can inform the PIC that a particular interrupt source be masked from delivery. However, this communication with the PIC is somewhat computationally expensive. The CPU most frequently controls interrupts by modifying a bit in the CPU’s flags register. If this bit is set to one, as can be accomplished using the STI instruction, then interrupts may be delivered to the CPU. If it is set to zero, as can be accomplished using the CLI instruction, then no maskable interrupts are delivered to the CPU.

3.1.3 Softirqs

A softirq is designated for execution by setting the corresponding bit in a bit field data structure which tracks pending softirqs. This bit field is surveyed after the execution of each hardirq. If any bits have been set, then a routine is called that is responsible for executing all pending softirqs. This routine begins by ensuring that no other hardirqs or bottom-halves are in the process of executing. It then takes a snapshot of the pending softirq bit field. All corresponding softirqs are then executed in ascending order. Therefore, the priority of a softirq is indicated by its position in the array of softirqs [2].

After one iteration through this snapshot, it is again compared to the pending softirq bit field. If any softirqs have been marked that were not executed in the previous iteration, then the snapshot is updated and the execution of softirqs is repeated. Otherwise, if any softirqs are still pending, then a kernel thread that executes softirqs in the same manner is marked as runnable and may be selected by the process scheduler..

3.1.4 Tasklets

Tasklets are a type of deferrable function [2]. The tasklet scheduling and execution function is implemented as a softirq. A tasklet is scheduled by prepending it to a list of pending tasklets. When the softirq scheduler decides to execute the tasklet softirq, the tasklet softirq first takes a snapshot of the list of pending tasklets and empties it. This list is surveyed in order. Each pending tasklet that is not concurrently running is executed. If a pending tasklet is currently running, then it's execution is skipped, it is reinserted into the pending tasklet list, and tasklet softirq is again marked for execution.

3.1.5 Bottom-Halves

Bottom-halves are a deprecated type of deferrable function [2]. The bottom-half scheduling and execution function is also implemented as a softirq. Although bottom-halves are considered obsolete, they are still used for some integral kernel tasks. Bottom-halves are implemented as tasklets. However, bottom-half tasklets are maintained separately from other tasklets. Bottom-halves also utilize concurrency mechanisms which ensure that all bottom-halve executions are serialized.

3.2 Latency

Latency is the amount of time that elapses between the assertion of a need to perform an action and its actual execution. There are many sources of latency in most operating systems. Concurrency control mechanisms create latency because they create a constraint that must be satisfied before execution may occur. The interaction of system computation policies can create unwanted latency. For example, the Linux policy of executing interrupt service routines as fast as possible affects the latency with which most other computations are executed. Although there are two types of latency, interrupt and scheduling, that have historically been treated separately, both are essentially equivalent under the unified scheduling model. This is due to the fact that the scheduling mechanisms for both interrupt service routines and threads have been combined.

3.2.1 Interrupt Latency

Interrupt latency is the amount of time between the assertion of an IRQ and the execution of the interrupt service routine that is associated with it. There are two significant causes of interrupt latency. First, interrupt concurrency control mechanisms block the delivery of interrupt signals. This prevents the operating system from acknowledging the presence of a pending interrupt signal. Second, because interrupts are executed “as fast as possible,” frequent high priority interrupts can postpone the execution of low priority interrupts.

3.2.2 Scheduling Latency

Scheduling latency refers to the amount of time between acknowledging that a thread should be executing and actually executing it. Scheduling latency ordinarily refers to only threads, as thread scheduling latency has historically been the focus of greatest scrutiny. However, the term can be expanded to include the latency in scheduling any type of computation.

There are many sources of scheduling latency. Interrupt latency can contribute to scheduling latency, as it prevents the operating system from receiving information that may indicate a need for a scheduling opportunity. An operating system's thread preemption capabilities are a major influence on thread scheduling latency. Older versions of Linux, for example, did not allow for preemption of a thread while it was executing in kernel code. Recent versions of Linux have reduced this restriction by allowing thread preemption at certain spots within the kernel.

3.3 The Scheduling Hierarchy

There are two types of components within the scheduling hierarchy, an example of which is given in Figure 3.1. First, each schedulable computational element is represented, possibly in multiple places. Second, entities known as "groups" are included. Groups are nodes within the hierarchy that are associated with a particular scheduling decision function (SDF). The SDF can be used to determine which of the group's children nodes should be chosen. These children nodes can be associated with either a computation or a group. A scheduling hierarchy is formed by constructing a hierarchy where groups are parent nodes of other groups. This hierarchy is traversed when the SDF of a parent node executes the SDF of a child node.

The scheduling hierarchy is employed by first calling the decision function that is associated with the group at the root of the hierarchy. This decision function, in turn, may call the decision functions associated with any of the groups that are members of the root group. Each of these children groups may also call the decision functions associated with their own member groups. Each parent group's decision function may then incorporate the decision of its member groups into its own decision-making process.

Ultimately, the decision of the root group is returned to the calling function. This function is then responsible for properly executing the computation that is chosen by the scheduling hierarchy. If no groups are present in the scheduling hierarchy, then a choice indicating that the hierarchy has no opinion about what to execute is returned.

The calling function must then decide what action to take.

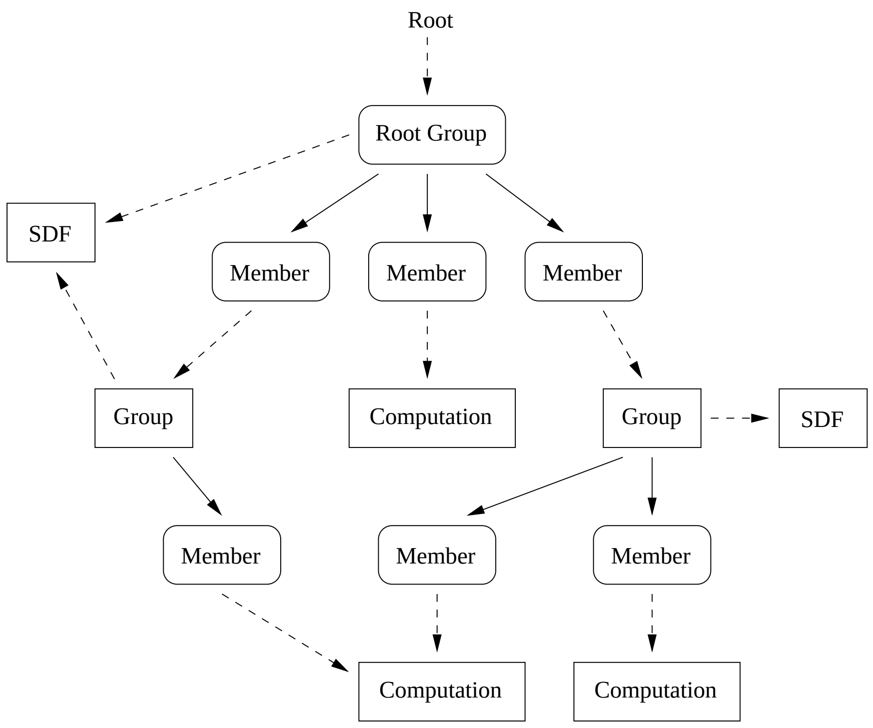

Figure 3.1: Example Scheduling Hierarchy

3.4 Scheduling Decision Functions

Scheduling decision functions (SDFs), at their core, are encapsulated routines that are responsible for choosing among a set of group members, which are described in Section 3.6. These routines often require accessory functions in order to operate correctly. Therefore, a structure is used to represent all the necessary elements of an SDF.

The SDF structure is shown in Program 3.1. Each SDF is given a distinctive name. This name is used to identify the SDF when creating a scheduling hierarchy. An SDF also contains pointers to three functions, of which only the decision function pointer is required to be non-NULL.

When the decision function is invoked, it decides which entity, if any, in its member set should be executed. This choice is then returned to the calling function. This decision function is only responsible for making a decision about which member to execute. It does not execute any of the members directly. Therefore, in order to create an SDF for a particular computation, the scheduling and execution semantics of the computation must be separated. Although groups may be members of other groups, they cannot be returned as the choice of a decision function. Members that represent computations are the only entities that should be returned, or chosen, by a decision function. Members that represent groups can be considered in a decision function by calling the member group's decision function. This decision function will return a member that represents a computation, which can then be considered in the current decision function.

The parameter setting function is used to communicate data that is used by the decision function. For example, if the SDF is a priority scheduler, then an application that wishes to set the priority of a particular entity in the group would call this function to accomplish such a task.

The member removal function is used to inform an SDF that one of its members has been removed. This function typically removes any state that the SDF was maintaining for the given member.

Each SDF maintains a reference count which indicates the number of groups that are currently associated with this SDF. An SDF cannot be unregistered if it is associated with any groups.

Program 3.1 Scheduler Decision Function (SDF) Data Structure

```
struct sdf {
    string name;
    pointer decision_function;
    pointer set_param_function;
    pointer remove_member_function;
    integer reference_count;
};
```

3.5 Groups

The scheduling hierarchy is created through the addition of an entity type known as a group. Groups are special entities that form internal nodes within the scheduling hierarchy and are responsible for directing the decision path that is taken through the scheduling hierarchy.

The structure of a group is shown in Program 3.2. Each group contains a numeric identifier for the group, as well as a distinctive name. It also contains a pointer to an SDF. This SDF determines the scheduling semantics that the group will impose on its members. The SDF can be unique to a particular group, or it can be shared among multiple groups.

Each group also possesses a pointer that can be used to store arbitrary data. This data can be used by the SDF in order to make scheduling decisions. Although a group's decision function may access static data, the use of dynamically allocated data allows for more than one group to utilize a particular SDF.

Each group maintains a reference count which indicates the number of groups to which it is a member. A group cannot be destroyed if it is a member of one or more other groups.

Finally, each group is associated with a list of members. This list is automatically maintained and updated by the facilities provided by the unified scheduling model. A group's SDF may use this list directly in its decision function. Alternatively, it may be more efficient for an SDF to maintain separate data structures related to each member. In this case, an SDF's member removal function should be utilized to remove and deallocate any data that is related to a particular member but is not stored within the member itself.

3.6 Group Members

Each group in the scheduling hierarchy contains a list of members. A group's SDF is responsible for choosing which of the members should be chosen for execution.

The structure of a member is shown in Program 3.3. It contains a pointer to the

Program 3.2 Group Data Structure

```
struct group {
    integer group_identifier;
    string name;
    pointer sdf;
    pointer data;
    integer reference_count;
    list members;
};
```

entity that this member represents and an indicator as to the type of this entity. These fields are used by a group's SDF to determine how to interpret a particular member.

If the entity type indicates a group, then the entity is interpreted as a pointer to another group. The SDF typically handles this entity type by calling the SDF associated with this member group. The choice by this group's SDF can then be considered for execution by the parent SDF.

Members also contain a pointer to data that can be considered during the execution of group's SDF, as described in Section 3.4. Although the information contained in this pointer can be considered by the group's SDF, the SDF is not limited to considering only this data.

Program 3.3 Member Data Structure

```
struct member {
    pointer entity;
    integer entity_type;
    pointer data;
};
```

3.7 Group Operations

The unified scheduling model provides a flexible interface that exposes every aspect of computation control to the user. A functioning scheduling hierarchy first requires that one or more groups be created. The hierarchical structure is formed among multiple groups by having each group "join" one or more groups. Finally, computations can also join one or more groups in order to be placed under the scheduling control of various decision functions.

3.7.1 Group Creation

Before a scheduling hierarchy can be formed, individual groups must be created. The pseudo-code for creating a group is given in Program 3.4.

The group creation function accepts a group name and an SDF identifier. First, the group name is verified to be unique, as no two groups can possess the same name. Next, the given SDF's existence is confirmed. After these checks are made, a new group is allocated and the SDF is associated with it. The SDF's reference count is incremented in order to indicate the new group association. The new group's reference count is initialized to zero.

The new group is then stored in the global group data structure. This data structure is simply a storage system for the groups that are present in the system. It does not have any semantic relationship to the group hierarchy.

Finally, if no other groups exist, then the new group becomes the root of the system scheduling hierarchy. The new group's identifier is then returned to the calling function.

3.7.2 Joining a Group

The fundamental method for creating a group hierarchy is through joining groups and computations together. The pseudo-code for joining a group is given in Program 3.5.

The group join function accepts a group identifier, a member identifier, and a member type. The group identifier refers to the group that the member wishes to join. The member identifier is a unique label that is specific to a given member type. For example, if the member type is a process, then the member identifier is equal to the process identifier. Since groups can also be members, then if the member type is a group, then the member identifier is equal to the group identifier. Since members are identified under the unified scheduling model by both their member identifier and their member type, duplicate identifiers between members of different types is allowed. Duplicate

Program 3.4 Pseudo-Code for Creating a Group

```
int create_group(group_name, sdf) {
    if (group_exists(group_name)) {
        return error;
    }
    if (sdf_does_not_exist(sdf) {
        return error;
    }
    new_group = allocate_new_group(group_name);
    new_group.sdf = sdf;
    sdf.reference_count++;
    new_group.reference_count = 0;
    store_group_in_global_data_structure(new_group);
    if (no_top_level_group_exists()) {
        top_level_group = new_group;
    }
    return new_group.group_identifier;
}
```

identifiers assigned to members of the same computation type must be prevented by the system and is outside the scope of the unified scheduling model.

The group join function first confirms that the group identifier refers to a valid group. Next, it verifies that the computation or group that is specified by the given member identifier and member type pair exists. If this test succeeds, then an existing membership in this group by the candidate member is tested. Since the ability to uniquely identify any given member in a given group's member list is required, attempts to have a specific member join a group more than once are prohibited.

If the previous tests are passed, then a new member structure is allocated. This new member is then appended to the specified group's member list. Furthermore, if the member is a process, then the group is appended to the process's group list as well. Otherwise, if the member is a group, then its reference count is incremented. Success is indicated to the calling function by returning the value zero.

3.7.3 Leaving a Group

The scheduling hierarchy allows for the removal of members from groups. The pseudo-code for leaving a group is given in Program 3.6. The group leave function accepts a

Program 3.5 Pseudo-Code for Joining a Group

```
int join_group(group, member_id, member_type) {
    if (group_does_not_exist(group)) {
        return error;
    }
    if (member_does_not_exist(member_id, member_type) {
        return error;
    }
    if (member_has_already_joined_this_group) {
        return error;
    }
    new_member = allocate_new_group_member(member_id,
                                           member_type);
    append_to_group's_member_list(group, new_member);
    if (member_type == process) {
        append_to_process's_group_list(group, new_member);
    } else if (member_type == group) {
        new_member.entity.reference_count++;
    }
    return 0;
}
```

group identifier that designates the group from which the member should leave as well as a member identifier and member type pair which identify the member that should leave the group.

After the existence of the given group is confirmed, the validity of the member is first checked by locating the given computation or group on the system and then verifying that it is a member of the specified group. If these tests are passed, then if a member removal function is present in the SDF that is associated with the group, it is executed. If this function returns an error, then the group leave function returns an error immediately.

Success of the SDF's member removal function is followed by removal of the member from the group's member list. Because of this, the SDF's member removal function should not remove a member from the group's member list. Instead, it should simply remove any state that the SDF was tracking that was related to the member. Furthermore, the group-specific data that is stored in the member's data pointer is deallocated by the group leave function and does not need to be freed by the SDF member removal function.

If the member that is leaving is a process, then the group is removed from the process's list of groups. Otherwise, if the member is a group, then its reference count is decremented. A value of zero is then returned to the caller.

Program 3.6 Pseudo-Code for Leaving a Group

```
int leave_group(group, member_id, member_type) {
    if (group_does_not_exist(group)) {
        return error;
    }
    if (member_does_not_exist(member_id, member_type) {
        return error;
    }
    if (member_is_not_a_member_of_this_group) {
        return error;
    }
    if (member_removal_function_exists(group.sdf)) {
        if (remove_member_function(group, member) != 0) {
            return error;
        }
    }
    remove_from_group's_member_list(group, member);
    deallocate(member.data);
    if (member_type == process) {
        remove_from_process's_group_list(group, new_member);
    } else if (member_type == group) {
        new_member.entity.reference_count--;
    }
    return 0;
}
```

3.7.4 Group Parameter-Setting

An SDF often needs data to be supplied by the user in order to make its decisions. This is accomplished through the group parameter-setting function. The pseudo-code for setting group parameters is given in Program 3.7.

The group parameter-setting function accepts several parameters. The group whose scheduling parameters are being modified must be given. A setting number, which indicates what type of information is being supplied, must also be specified. A member identifier and member type pair are accepted as well. These arguments indicate the member to which the scheduling parameters are applied. If the scheduling parameters are not specific to a particular member, then these arguments are ignored. Finally, a pointer to a parameter value, along with its size, is accepted.

The group parameter-setting function first verifies that the specified group exists. It then attempts to locate the member that is denoted by the member identifier and member type pair. The presence or absence of this member is noted and the parameter-setting function of the SDF that is associated with the group is executed. The return value of this function is then returned to the caller.

Program 3.7 Pseudo-Code for Group Parameter-Setting

```
int group_parameter_setting(group, setting, member_id,
                               member_type, value, size) {
    if (group_does_not_exist(group)) {
        return error;
    }
    if (member_does_not_exist(member_id, member_type) {
        member = NULL;
    } else {
        member = find_member(member_id, member_type);
    }
    return_value = set_param_function(group, member);
    return return_value;
}
```

3.7.5 Group Destruction

At some point during its lifetime a group may no longer be necessary. This group may then be purged from the system through the group destruction routine. The pseudo-code for destroying a group is given in Program 3.8.

The group destruction function accepts the name of the group whose destruction is requested. If no group with the given name exists, then an error is returned. A group may not be destroyed if it is in use. This is ascertained by first determining if the group has any members and then checking the group's reference count, which indicates if the group itself is a member of any other group.

Success of the previous tests certifies that the group can be destroyed. First, the

SDF's reference count is decremented to indicate the loss of a group association. Next, if the group that is being destroyed is also the root group in the scheduling hierarchy, then the root of the scheduling hierarchy is removed, and the hierarchy becomes empty. Finally, the group is removed from the global group storage structure and deallocated. A value of zero is returned to the calling function and indicates that the given group was successfully destroyed.

Program 3.8 Pseudo-Code for Destroying a Group

```
int destroy_group(group_name) {
    if (group_does_not_exist(group_name)) {
        return error;
    }
    if (group_member_list_is_not_empty(group)) {
        return error;
    }
    if (group.reference_count > 0) {
        return error;
    }
    group.sdf.reference_count--;
    if (group_is_top_level_group(group)) {
        top_level_group = NULL;
    }
    remove_group_from_global_data_structure(group);
    deallocate(group);
    return 0;
}
```

3.7.6 Atomic Hierarchy Creation

It is often not adequate to sequentially create a hierarchy through a series of group joining. Therefore, the ability to specify a hierarchy atomically has been added. This is accomplished by submitting an array of structures known as atomic hierarchy nodes. Each of these nodes specifies the relationship between a parent group and a set of child groups. This function only specifies a hierarchical relationship and does not actually create the groups that are referenced by each atomic hierarchy node.

The structure of an atomic hierarchy node is shown in Program 3.9. First, the identifier of a particular group is specified as the parent group. Next, the number of children that this group has is given. Finally, the structure contains a pointer to an array of group identifiers. The groups corresponding to these identifiers are each added as members of the parent group atomically.

Program 3.9 Atomic Hierarchy Specification Node

```
struct atomic_hierarchy_node {
    integer parent_group_identifier;
    integer num_children_groups;
    pointer children_group_id_array;
};
```

3.8 Writing an SDF

Because each SDF is specific to the policy that a system designer wishes to implement, very few generalizations can be made of all SDFs. However, there is a limited set of rules that each SDF must obey, and most SDFs can be constructed in a similar fashion.

3.8.1 The Decision Function

The decision function is the nucleus of an SDF structure. An example of what a conventional decision function might do is given in Program 3.10. All decision functions accept the same argument list. A pointer to the thread that possesses context is given as well as an identifier of the CPU on which the scheduling hierarchy is being executed. The group with which the decision function is associated is also supplied. This allows the decision function to access the group's internal data, such as its member list.

The example decision function initially sets its choice to be PASS. This indicates that the decision function has no opinion about what computation should be run and is willing to pass the decision to another group.

The decision function then surveys the members of the group. First, it determines if a member is ready, meaning that is capable of being executed at the present moment. Members that are processes, for example, might be in a blocked state. If the member is ready, then the scheduling data associated with the member is considered. If this data indicates that the member should be chosen, then the member type is inspected. If the member is a group, then the member group's decision function is executed and its choice is considered. If the member does not represent “no opinion,” then it is selected as a candidate for execution. Once all the members have been surveyed, the result of the decision function can then be scrutinized, or it can simply be returned as the final choice, as is done in the example shown in Program 3.10.

Program 3.10 Pseudo-Code for a Decision Function

```
member decision_function(previous_task, this_cpu, group) {
     choice = PASS;
     for (each_member_in_group_member_list(group)) {
          if (member_is_ready(member)) {
               consider_member_data(member.data);
               if (this_member_is_better(member, choice)) {
                    if (member.entity_type == group) {
                         member = run_decision_func(previous_task,
                                                    this_cpu,
                                                    member.entity);
                    }
                    if (member != PASS)
                         choice = member;
               }
          }
     }
     return choice;
}
```

3.8.2 The Parameter-Setting Function

The parameter-setting function is an SDF accessory routine. Most parameter-setting functions mimic the functionality shown in Program 3.11. A parameter-setting function must accept a specific set of arguments. The group to which the value is being applied is indicated. The *setting* argument denotes which scheduling parameter is being specified. This allows the parameter-setting function to multiplex the communication of multiple types of parameters to a decision function. The next argument specifies the member to which the parameter pertains. Some types of parameters may not be specific to a particular member. Therefore, this argument can optionally be ignored. The final two arguments contain a pointer to the value of the parameter and the size of the value.

A parameter-setting function first consults the *setting* argument. A course of action is then taken based on its contents. Program 3.11 displays a parameter-setting function with two different paths. The application that is calling the parameter-setting function must have knowledge of what settings are available and what actions correspond to each setting. In the example, if *setting* has a value of SETTING_NUMBER_ONE, then the member argument is consulted, because the parameter associated with SETTING_NUMBER_ONE is defined to be member-specific. Next, if necessary, it allocates space to store the parameter in the member's data pointer. Finally, the given parameter is copied into this space. The second path, which is taken when *setting* has a value of SETTING_NUMBER_TWO, does not expect a parameter that is specific to a particular member. Therefore, it ignores the *member* argument and stores the value in the group's data pointer.

Program 3.11 Pseudo-Code for a Parameter-Setting Function

```
int set_param_function(group, setting, member, value, size) {
    switch(setting) {
        case SETTING_NUMBER_ONE:
            if (member == NULL) {
                return error;
            }
            if (member.data == NULL) {
                allocate(member.data, size);
            }
            copy(member.data, value, size);
            break;
        case SETTING_NUMBER_TWO:
            copy(group.data.variable_number_one, value, size);
            break;
        default:
            return error;
    }
    return 0;
}
```

3.8.3 The Member Removal Function

The member removal function is another SDF accessory routine. Since it is responsible for removing a member from the data structures that a group maintains, it is entirely dependent on the form of these data structures. An example of a member removal function is given in Program 3.12.

Member removal functions accept a member and the group which it is leaving. The example function iterates through the group's data structures looking for any data that refers to the member that is leaving the group. If data is found, then it is removed and deallocated. A return value of zero indicates success. If a non-zero value is returned, then the member is not allowed to leave the group. Such a situation might arise, for example, when a child thread wishes to leave a group but its parent thread is not ready for it to finish.

Program 3.12 Pseudo-Code for a Member Removal Function

```
int member_remove_function(group, member) {
    for (each_group_data_structure(group)) {
        if (data_relates_to_member(member)) {
            remove_data();
            deallocate_data();
        }
    }
    return 0;
}
```

3.8.4 SDF Operations

The creation of an SDF is not complete until the SDF data structure, which is shown in Program 3.1 on page 20, is specified and accessible. An SDF is useless unless it is associated with one or more groups within the scheduling hierarchy. In order for groups to associate themselves with a particular SDF, the SDF must first register itself with a global data structure that provides a mapping between SDFs and their unique names. SDF registration is done in the kernel module initialization routine.

The function that registers an SDF accepts four arguments. These arguments correspond to the first four fields in the SDF data structure. The registration function first verifies that the SDF's name is unique. It then allocates an SDF structure, initializes it with the supplied arguments, and sets the SDF reference count to zero.

If an SDF is no longer being used, then it can be removed from the global SDF storage structure. This is done by calling the SDF unregister function. This function accepts the name of the SDF that is to be removed. This function returns an error if an SDF with the given name does not exist or if the SDF reference count is greater than zero.

3.9 Computational Elements

The unified nature of the scheduling model establishes the ability of the scheduling hierarchy to consider a wide range of computations. These computations, which were previously scheduled and executed separately, are now controlled under a single model.

3.9.1 Processes

Processes were the first computations to be placed under control of the unified scheduling model. There are three reasons for this. First, process scheduling has the greatest impact on system performance. Second, the semantics that system designers most commonly desire are primarily related to process scheduling. Finally, the nature of processes makes the separation of their scheduling and execution mechanisms trivial. Such a natural separation provided a good starting place for implementation of the unified scheduling model.

Processes are unique among system computations in two ways. First, they possess their own context. As a result, in order to execute a process that does not already have CPU context the current thread of execution must be temporarily abandoned. This interruption of the current thread creates a natural boundary between scheduling and execution. Second, processes can be created and destroyed. This unique attribute requires that special consideration be given to processes at the time of their destruction. Given these distinguishing properties, the integration of processes with the unified scheduling model has unique properties.

Under the unified scheduling model, the standard Linux scheduler is not expressed as an SDF. Instead, a placeholder member type is created in order to represent the standard Linux scheduler and all the processes that have not been explicitly placed under control of the scheduling hierarchy. This is done in order to simplify the control that system designers must exert. Encapsulating the standard Linux scheduler into an SDF and associating it with one or more groups would require the system designer to explicitly state the group membership for every process that is created. System designers would rather concentrate on a select group of processes that are needed in an application for which specialized scheduling semantics are desired and treat all other processes on the system as a single entity. The placeholder allows for this. The semantic equivalent of the scheduling hierarchy shown in Figure 3.2 is displayed in Figure 3.3. In the actual system, the scheduling hierarchy contains a placeholder for the standard Linux scheduler. If the scheduling hierarchy does not return the placeholder as its choice, the chosen computation is executed as usual. However, when the scheduling hierarchy returns this placeholder as its choice for execution, the standard Linux scheduler is run and its choice is executed. This is semantically equivalent to implementing the standard Linux scheduler as an SDF, associating it with one or more groups in the hierarchy, and then having each process join these groups.

Although each process is implicitly placed under control of the standard Linux scheduler upon creation, a method is needed for preventing the standard Linux scheduler from selecting processes that are exclusively controlled by the scheduling hierarchy. This is accomplished by adding a particular flag, SCHED_NOSELECT, to each process. The standard Linux scheduler is modified slightly to check this flag. If it is set, then the process is ignored.

Because processes can be dynamically created and destroyed, precautions must be taken in order to maintain a consistent scheduling hierarchy. Each process maintains a list of groups to which it is a member. When a process is created through the **fork** system call, this list is initialized as empty. Each time a process joins a group, as discussed in Section 3.7.2, the group's identifier is added to the list. Likewise, each time a process leaves a group, as discussed in Section 3.7.3, the group's identifier is removed from the list. When a process is destroyed through the **exit** system call, it is removed from each of the groups in its group list.

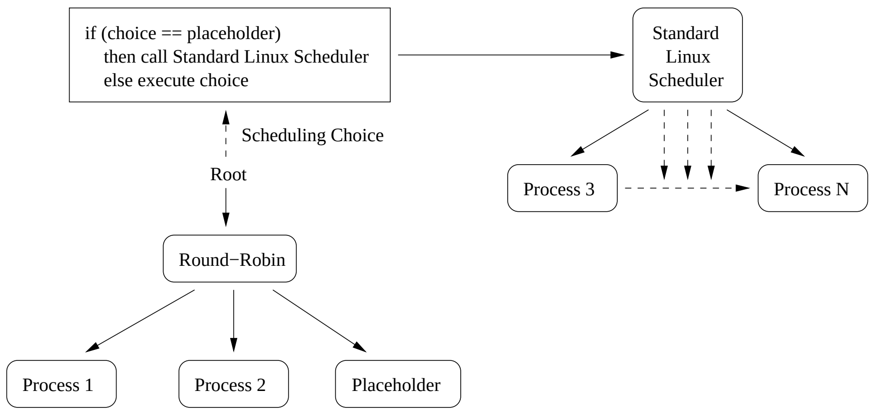

Figure 3.2: Structure of a Hierarchy that Incorporates the Standard Linux Scheduler Placeholder

3.9.2 Hardirqs

Hardirqs have become a controllable computation under the unified scheduling model. This has been accomplished by handling interrupt concurrency control and scheduling in software rather than hardware. Software control allows for much more precise control of hardirqs and more complex interrupt processing decisions. System designers are no longer limited by the constraints that hardware mechanisms place on hardirqs. In order to place hardirqs under the unified scheduling model, their scheduling and execution semantics needed to be separated and explicitly stated. This task differs from other computations in that much of these semantics are located in hardware mechanisms.

Software control of interrupt concurrency is handled by an SDF designed specifically for hardirqs. This SDF can make decisions based on a complex set of semantics rather than the simple priority mechanism that is provided by traditional interrupt hardware. When a critical section attempts to disable interrupts at the hardware level, this fact is noted in a software data structure but not actually carried out on the hard-

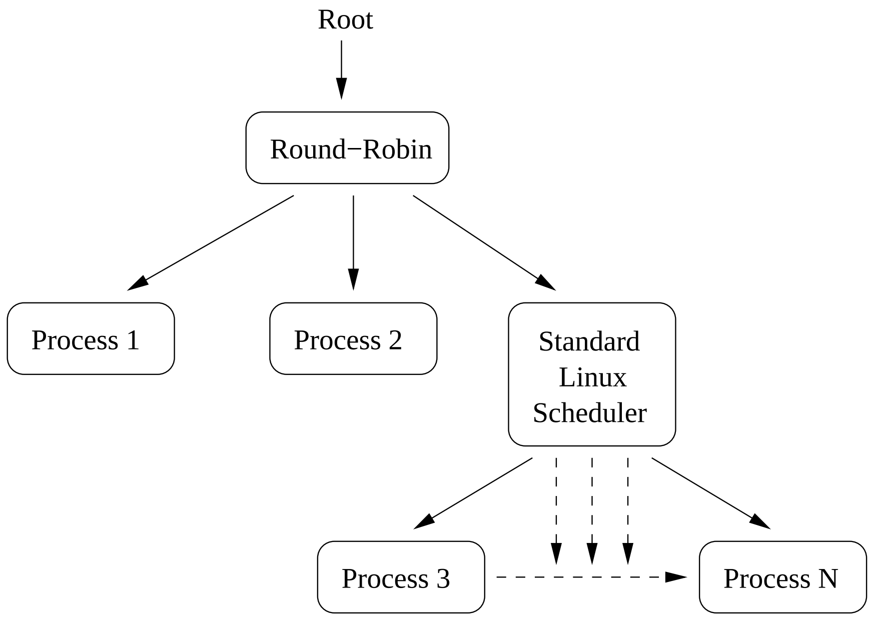

Figure 3.3: Semantic Equivalent of the Standard Linux Scheduler Placeholder

ware. When a subsequent interrupt occurs, it is marked in a separate data structure and control is then transferred to the scheduling hierarchy. Based on the available data structures, the scheduling hierarchy can then decide whether to run the corresponding hardirq or postpone its execution until a later time. The scheduling hierarchy is also called when exiting a critical section.

The semantics of critical sections have been extended further to take advantage of this new method of interrupt processing. Upon entering a critical section, it is no longer necessary to signal that all interrupts should be disabled. Instead, only those interrupt handlers which pose a concurrency conflict need to be disabled.

The introduction of a scheduling hierarchy adds significant flexibility to interrupt processing as well as the potential accuracy of real-time tasks. The scheduling hierarchy can choose to execute real-time tasks regardless of conditions that might have prevented their execution under standard Linux interrupt processing.

3.9.2.1 Concurrency Control

There are several concurrency control mechanisms in the Linux kernel. Each of these mechanisms manipulates various data structures, which are described in Table 3.1. The information contained in these data structures must be considered by an SDF that schedules hardirqs in order to prevent concurrency conflicts.

There are two fundamental functions, *local_irq_disable* and *local_irq_enable*, which form the basis of Linux interrupt concurrency control. *Local_irq_disable* is a wrapper around the CLI processor instruction, which disables interrupts by clearing the interrupt flag that is located in the processor. *Local_irq_enable* is a wrapper around the STI processor instruction, which enables this flag.

These functions have been modified such that they no longer act upon the processor interrupt flag. Instead, the semantics of the CLI and STI instructions have been implemented in software to operate on a software interrupt flag variable. The *local_irq_disable* function sets the software interrupt flag to zero and also disables task preemption. This disabling is necessary in order to ensure the semantics of Linux task preemption, which assume that asynchronous preemption is not possible when interrupts are disabled. The *local_irq_enable* function first determines if hardirqs are already enabled by checking the state of the software interrupt flag. Since the STI instruction has no effect if interrupts are already enabled, the function immediately exits if this is the case. Otherwise, the software interrupt flag is set to one and task preemption is enabled, as is consistent with standard Linux task preemption semantics. The list of pending interrupts is then surveyed. If hardirqs have occurred but have not been serviced while the software interrupt flag was disabled, then the local_irq_enable function will call the unified scheduling hierarchy in order for the hardirq SDF to have an opportunity to execute any pending hardirqs. This follows the semantics that would normally occur under the hardware enable and disable mechanism used by standard Linux.

In order to extend concurrency control beyond the semantics provided by the global software interrupt flag, it is necessary to find a method for determining which parts of the code conflict with hardirqs. Linux already provides a good starting place in its spin-

Hardware Interrupt Flag	This flag should be disabled during access to any data that is used by the small irq handler.
Software Interrupt Flag	This flag must be disabled during access to any data that is used by a hardirq. This flag is not sufficient for protecting data that is accessed by hardirqs that ignore this flag.
Preemption Count	This counter variable is associated with each process. Whenever it is equal to zero, the current process may be preempted by another process. Asynchronous preemption cannot occur while an interrupt flag is disabled.
Spin Locks Read / Write Locks	Locks protect data that can be accessed concurrently by multiple CPUs. When a lock is acquired by a process, its preemption count is incremented in order to prevent the process from being preempted while holding a lock.
Bottom-Half Flag	This flag synchronizes access to bottom half data structures.
Interrupt-Specific Flags	A specific flag is designated for each hardirq and can be used as a replacement for the software interrupt flag.

Table 3.1: Concurrency Control Data Structures

lock functionality. Code that contains concurrency conflicts with hardirqs is typically protected by spinlock functions that also enable and disable the hardware interrupt flag. These functions provide an easy mechanism for extending the software interrupt logic that has been introduced.

Once the spinlocks that protect interrupt handler code have been identified, they can be associated with a particular interrupt source. Subsequent attempts to obtain the spinlock and disable the global interrupt flag instead disable only the associated interrupt source. This allows non-conflicting interrupts that would normally be blocked by the global interrupt flag to be executed concurrently.

3.9.2.2 Interrupt Execution Flow

When an interrupt source is asserted at the hardware level, the CPU disables interrupts by setting the processor interrupt flag to zero. The current thread of execution is interrupted, and control is then passed to a small interrupt handler, as illustrated in Figure 3.4. This function communicates with the PIC. It first masks the interrupt source that occurred and then sends an end-of-interrupt signal. This signal allows the PIC to assert interrupts that have a lower priority at the hardware level than the interrupt that just occurred. However, these interrupts will not be acknowledged by the processor until the hardware interrupt flag is re-enabled.

The hardirq corresponding to the interrupt source is then appended to the list of pending hardirqs. The hardware interrupt flag is enabled, and the scheduling hierarchy is then given an opportunity to run. If the scheduling hierarchy chooses a context-borrowing computation, such as a hardirq or a softirq, then it is immediately executed. The scheduling hierarchy is repeatedly called until it chooses a computation that cannot be immediately executed. It then returns, and the interrupted context is restored.

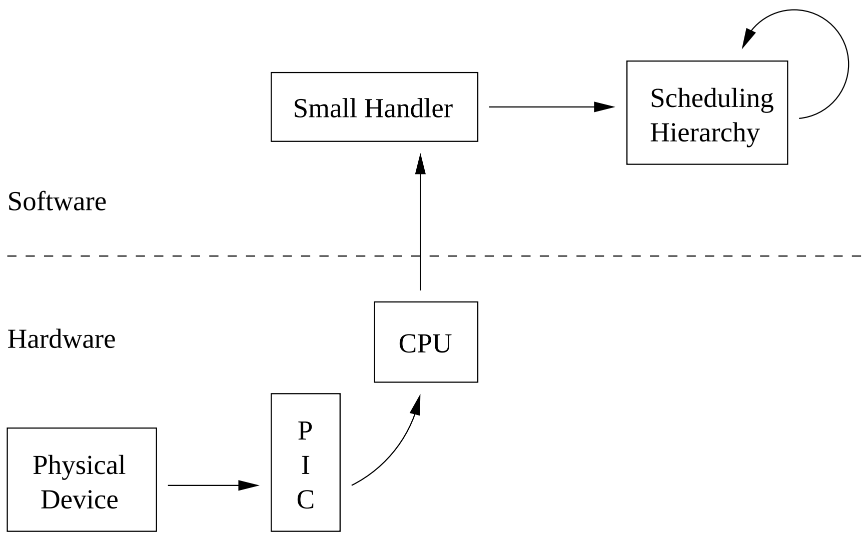

Figure 3.4: Interrupt Execution Flow

3.9.2.3 SDF

Although hardirqs can be incorporated into any SDF, an example set of SDFs that allow for simple hardirq functionality has been created. The hardirq SDFs are responsible for examining the current state of the hardirq data structures, including concurrency control variables, and selecting the optimal hardirq for execution, if any. Their current implementation could easily be modified to suit the requirements of a particular system, similar to a process scheduler. There are two SDF that have been created to control hardirqs.

The first SDF is responsible for controlling execution of the timer interrupt. On most commercial, off the shelf (COTS) systems, this will be the only real-time interrupt source, as all real-time tasks are signaled by a timer chip interrupt. The SDF first tests whether the timer interrupt is either specifically blocked or is currently running. If either of these conditions is true, then a member called RETURN is chosen. This special type of member indicates that no new computation should be chosen for execution. Instead, the currently running computation should be finished. Since the timer interrupt is the sole real-time interrupt source in the system, it should not be interrupted by any other computations. If the timer interrupt is pending and is not specifically blocked, then the decision function selects it. Otherwise, PASS is returned. This SDF violates the interrupt handling semantics present in standard Linux, as it allows an interrupt to be asserted and executed while interrupts are disabled from standard Linux's point of view through the software interrupt flag. In order to correctly deal with this semantic change, all code that conflicts with the timer hardirq is protected by specifically blocking the timer hardirq.

The other hardirq SDF implements a modified version of the interrupt priority scheme that is present in the interrupt hardware. If the software interrupt flag is enabled, then the list of pending hardirqs is surveyed to determine if any interrupt sources have been asserted. The corresponding hardirqs are then chosen based on their numeric priority, which is determined by the interrupt request line identifier that is associated with each interrupt source. One modification to the standard Linux interrupt scheduling algorithm is the consideration of interrupt-specific concurrency control. Although the global software interrupt flag might be enabled, a specific interrupt source might be disabled. In this case, its execution would be postponed, and another hardirq might be chosen. If no hardirqs are selected, then PASS is returned. If the software interrupt flag is disabled, then RETURN is chosen.

The two hardirq SDFs, along with the softirq SDF that is discussed in Section 3.9.3, are each associated with a group. These three groups are placed into the scheduling hierarchy as children of a top-level group. This arrangement is displayed in Figure 3.5. The decision function of this top-level group simply considers its members in order. For example, if its first member is a group, then the member group's decision function is executed. If the choice made by this member group is not PASS, then it is returned. Otherwise, the second member is considered, and so forth. Thus, the timer hardirq group is placed as the first member of the top-level group, the general hardirq group is second, and the softirq group is third. Additional groups, such as a process scheduler, can then be placed into the hierarchy by joining the top-level group and adjusting their placement.

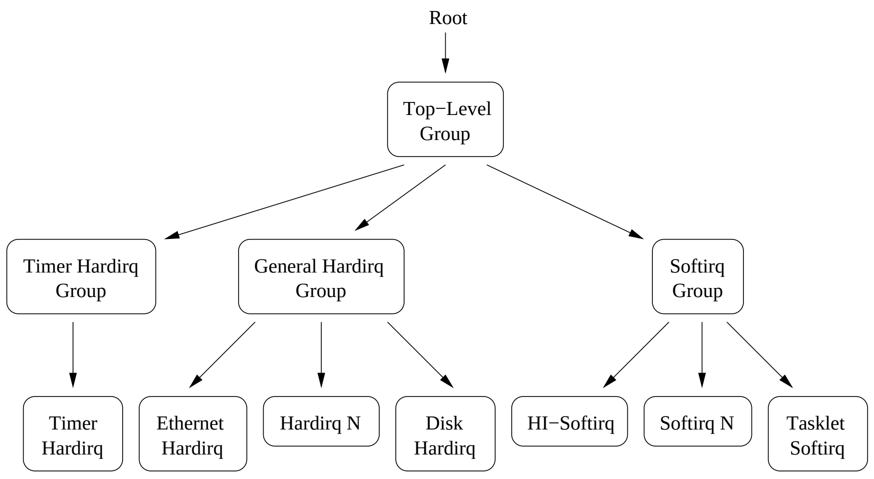

Figure 3.5: IRQ Scheduling Hierarchy

The timer hardirq has been modified in order to increase real-time scheduling accuracy of Linux timer computations. Standard Linux assumes that all kernel timers are of equal importance, and these kernel timers are executed in the timer bottom-half. Timers with real-time constraints are thus executed in the timer bottom-half along with all other kernel timers. This is not sufficient for real-time computations, as bottom-halves have worse response time and lower priority than hardirqs. Since timer computations have not yet been placed under control of the unified scheduling model, this problem was fixed by adding a specific flag field in real-time timers that allows them to be executed directly in the timer hardirq. While this adds greater response accuracy to real-time kernel timers, it places a significant restriction on them. Since the decision function can choose the timer hardirq even when the software interrupt flag indicates that interrupts are disabled, these kernel timers cannot have concurrency conflicts with code that uses interrupt disabling to control concurrent access to data. Any conflicting code must be protected by specifically disabling the timer hardirq.

When a hardirq is selected for execution by the scheduling hierachy, it is removed from the list of pending hardirqs. The software interrupt flag and preemption are both disabled in order to replicate the environment that would normally be present under standard Linux. The hardirq execution function is then called.

3.9.3 Softirqs

These computations are much easier to put under a unified scheduling model because they exist entirely in software and are already controlled by a software algorithm. However, the algorithm that is used to control softirqs in Linux does not easily lend itself to the unified scheduling model.

The softirq scheduling algorithm is reliant upon the softirq execution mechanism. Because of this, there are two options for encapsulating it as an SDF. State variables can be created to track which softirqs are actually executed and at what time they are executed, or the softirq scheduling algorithm can be modified to remove its dependence on execution information. This section describes the latter approach.

The softirq decision function assumes that it is only called after the hardirq decision functions have expressed no opinion. This is a slight optimization, which prevents the softirq decision function from having to check the software interrupt flag. Thus, the softirq decision function first ensures that no other hardirqs or bottom-halves are executing. If this check fails, then RETURN is returned. Otherwise, the highest priority pending softirq is determined by surveying the pending softirq bit field, and the corresponding softirq member is returned. If no softirqs are pending, then PASS is returned.

The softirq execution function is called when the hierarchy chooses to execute a softirq. It first clears the bit associated with the chosen softirq in the pending softirq bit field. It then executes the softirq and returns.

Process	Computations of this type are executed using a context-switch.
Hardirq	Computations of this type are executed by calling the computation function directly.
Softirq	Computations of this type are executed by calling the computation function directly.
Group	Members that are associated with this computation type are not chosen by the scheduling hierarchy. Instead, decision functions can directly execute the decision function associated with a member of this type.
Nothing	This computation type signifies a placeholder for the Linux scheduler. It signifies that the scheduling hierarchy has chosen nothing but that another scheduler may make a choice.
Pass	This computation type can be returned by decision functions to assert that they have no opinion about what to choose.
Return	This computation type can be chosen by a decision function in order to signify that no choice should be made by the hierarchy and that the current thread should continue execution by returning from the calling function.

Table 3.2: Computation Types

3.9.4 Adding New Computational Elements

Other types of computation can be placed under the unified scheduling model. The first step in this process is the separation of the existing scheduling and execution code. Once this is achieved, a data structure needs to be associated with the new computation type. This can be done by either adopting a structure that already exists in standard Linux, modifying a structure that already exists in standard Linux, or creating a new structure. The only requirement imposed on this structure is that it must contain a unique identifier for the computation. The internal functions of the unified scheduling model must then be modified to use this identifier in locating computations of the new type. Also, a new computation type identifier must be designated for the new type, and the the internal methods must be modified to acknowledge it. The computation types that are currently defined are given in Table 3.2. Finally, code that calls the scheduling hierarchy must be modified to execute the member if it is chosen.

Chapter 4

Evaluation

The most effective method for evaluating the unified scheduling model is to implement both existing and desired programming models using this new framework. A programming model consists of underlying operating system support, an API library, and user-level applications.

4.1 Explicit Plan Programming Model

KURT-Linux provides an example of an existing programming model [17]. KURT-Linux adds real-time thread scheduling capabilities to the operating system, implements an API library, and provides example applications which exercise the system's abilities. The explicit plan SDF is an implementation of the KURT-Linux programming model that is built upon the facilities provided by the unified scheduling model. It demonstrates how an existing, useful programming model can be adapted to work under this new framework.

The KURT-Linux programming model allows processes to be explicitly assigned intervals of execution. This is accomplished by providing facilities for processes to register themselves as real-time processes, set a desired scheduling mode, submit a schedule, and suspend their execution. Internally, KURT-Linux maintains a listing of real-time processes along with their real-time attributes. Upon submission of a schedule, Linux timers are created, utilizing the timing resolution created by the UTIME project, which are used to induce scheduling opportunities at the correct moment. When one of these timers expires, the start or finish of a real-time process interval is noted, future timers may be created, and the need for a scheduling opportunity is signified. KURT-Linux modifies the Linux scheduler by placing a call to the KURT-Linux scheduler in front of the normal Linux scheduling algorithm. If the KURT-Linux scheduler chooses a process for execution, then a context switch is immediately made to the chosen process. Otherwise, the choice is made by the Linux scheduling algorithm.

The KURT-Linux scheduler considers several sources of information when choosing a process, if any, for execution. It surveys whether the start or end of a process interval has occurred based on the data provided by any expired Linux timers. It also considers the scheduling mode that a process has specified. For example, a process may choose to execute only within its assigned intervals, or it may choose to also execute outside of assigned intervals on a best-effort basis.

Much of the functionality that is provided by KURT-Linux can be reproduced and enhanced by utilizing the unified scheduling model. Special functions for marking a process as real-time are no longer necessary. Instead, processes can simply join a group that is associated with the explicit plan SDF. The semantics produced by various scheduling modes can be replicated and extended by manipulating a process's group membership. For example, a process can be a member of a group that is associated with the explicit plan SDF in order to be executed during particular intervals. It can also join other groups, such as those associated with the standard Linux scheduler, in order to also be considered under a best-effort algorithm.

4.1.1 Explicit Plan SDF

The explicit plan SDF chooses members based on their assigned intervals of execution. The interval data structure, which is shown in Program 4.1, specifies a duration of time during which a particular member is granted exclusive access to the CPU. It consists of a member identifier and type, which denote the member to which the interval pertains. It also contains beginning and ending times. During this interval, the specified member is granted exclusive access to the CPU that is specified in the processor field. If the processor field is less than zero, then the member is granted access to any available CPU. Finally, it contains the periodicity of the interval and the number of times that it should occur. Each time the interval expires, the period time is added to the beginning time of the interval and the instances field, which represents the lifetime of the interval, is decremented. This is repeated until the instances field is equal to zero. If the number of instances is set to a value less than zero, then the interval is repeated indefinitely.

Program 4.1 Interval Specification Data Structure

```
struct interval {
    integer member_id;
    integer member_type;
    time begin;
    time end;
    integer processor;
    integer instances;
    time period;
};
```

4.1.1.1 Decision Function

The explicit plan decision function, which is displayed in Program 4.2, selects members based on the list of intervals, updates expired intervals, and programs timers to ensure accurate scheduling. The decision function consists of a conditional loop that repeats as long as the schedule contains intervals. Only the first interval in the schedule is considered during each iteration of the loop, because the intervals within the schedule are sorted. Various tests are performed on this interval in order to decide what action to take. If the interval has not been suspended, the interval can be executed on the current processor, the current time lies within the interval, and the member is ready to execute, then the member associated with the interval is selected for execution and a timer to signal the interval's end time is created. The loop then exits. Otherwise, If the first interval on the list has expired, then its beginning and ending times are updated based on its periodicity and remaining lifetime. The scheduling loop is then repeated in order to account for the newly updated interval. Finally, if none of the previous checks have succeeded, then a timer for the interval is programmed and the decision function returns without having an opinion about what member should be chosen.

This decision function actually improves the functionality of the KURT-Linux programming model in that it allows processes to block and resume within the same interval. Furthermore, whereas the KURT-Linux programming model only incorporates process scheduling, the explicit plan scheduler can assign intervals to any type of computation. Finally, since intervals can be assigned to groups, complex semantics can be implemented by assigning intervals to a group of computations and using a subordinate scheduler to determine which computation is chosen during the interval.

The overhead of this decision function is shown in Figure 4.1. This test was performed over a period of 10,000 seconds on a computer system with a 200 MHz Pentium Pro processor, generating approximately 100 million data points. System load was generated by cyclically executing a two-process kernel compilation on the local hard disk through a secure shell connection.

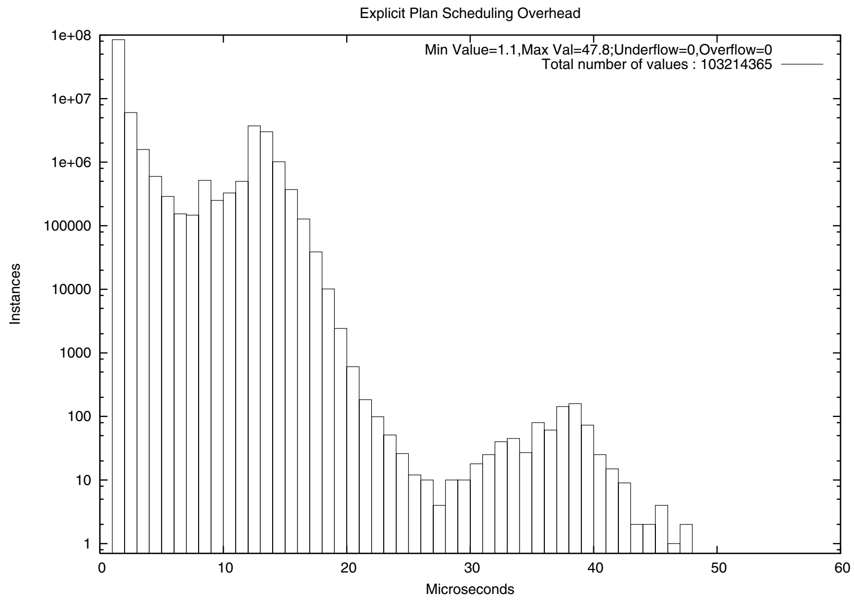

Figure 4.1: Explicit Plan Decision Function Overhead

Program 4.2 Pseudo-Code for the Explicit Plan Decision Function

```
member explicit_plan_decision(previous_task, this_cpu, group) {
    choice = PASS;
    schedule = group.data.schedule;
    while (not_empty(schedule)) {
        current_interval = first_interval(schedule);
        if (not_suspended(current_interval) AND
            ((current_interval.processor < 0) OR
             (current_interval.processor == this_cpu)) AND
            current_time_is_within_interval(current_interval)) {
            if (member_is_ready(current_interval.member)) {
                choice = current_interval.member;
            }
            set_next_timer(current_interval);
            break;
        } else if (interval_has_expired(current_interval)) {
            update_expired_interval(current_interval);

        } else {
            set_next_timer(current_interval);
            break;
        }
    }
    if (choice.entity_type == group) {
        choice = run_decision_func(previous_task, this_cpu,
                                    member.entity);
    }
    return choice;
}
```

4.1.1.2 Parameter-Setting Function

There are three specialized API functions which are provided by the explicit plan SDF. Each of these functions is implemented as a unique setting within the parameter-setting function of the SDF, which is shown in Program 4.3.

The *SUBMIT_EXPLICIT_PLAN* setting is used to submit a schedule, which is structured as an array of intervals. The validity of the schedule is first confirmed. This prevents errors such as overlapping intervals. The schedule is then inserted into the group's list of intervals. A rescheduling opportunity is signaled, and the function exits.

The *SUBMIT_EXPLICIT_PLAN_WAIT* setting is also used to submit a schedule. However, it blocks the calling process until the schedule is finished or a signal is delivered. This is accomplished by placing the calling process on the group's explicit plan wait queue and sleeping. When the schedule finishes, processes on the wait queue are awakened.

The *SUSPEND_INTERVAL* setting allows a computation to suspend its execution during any intervals in which it is currently executing. This prevents the computation from executing until its next assigned interval. Suspending is useful for computations that wish to synchronize themselves with the explicit plan. It allows them to control exactly what work will be executed during a given interval.

This setting is unique in that it can be used to suspend a process that is not a member of the group. The motivation for this functionality is that it allows a set of child processes to suspend themselves, and their parent process can then submit a schedule for its children and wait for it to complete. The suspend setting begins by determining if a particular member has specified to be suspended. If one has, then if the member refers to a process, it is prevented from being executed by the standard Linux process scheduler. Furthermore, the intervals associated with the member are marked as suspended. This prevents the decision function from considering them. If no member has been specified to be suspended, then the current process is prevented from being chosen by the standard Linux process scheduler. Finally, a rescheduling opportunity is requested and the function exits.

Program 4.3 Pseudo-Code for the Explicit Plan Parameter-Setting Function

```
int explicit_plan_set_param(group, setting, member, value, size) {
    return_value = 0;
    switch(setting) {
        case SUBMIT_EXPLICIT_PLAN:
            return_value = verify_plan(group, value, size);
            if (return_value != 0) {
                break;
            }
            insert_plan(group, value, size);
            current.need_resched = 1;
            break;
        case SUBMIT_EXPLICIT_PLAN_WAIT:
            return_value = verify_plan(group, value, size);
            if (return_value != 0) {
                break;
            }
            insert_plan(group, value, size);
            interruptible_sleep_on(explicit_plan_wait_queue);
            if (signal_pending(current)) {
                return_value = -EINTR;
            }
            break;
        case SUSPEND_INTERVAL:
            if (member != NULL) {
                if (member.type == process) {
                    member.entity.policy |= SCHED_NOSELECT;
                }
                return_value = interval_suspend(member);
            } else {
                current.policy |= SCHED_NOSELECT;
                return_value = 0;
            }
            current.need_resched = 1;
            break;
        default:
            return_value = error;
    }
    return return_value;
}
```

4.1.1.3 Member Removal Function

The explicit plan member removal function shown in Program 4.4 is called whenever a member is leaving the group. This is done in order to remove any of the intervals remaining in the schedule that are associated with the leaving member.

Program 4.4 Pseudo-Code for the Explicit Plan Member Removal Function

```
int explicit_plan_member_remove(group, member) {
    schedule = group.data.schedule;
    for (each_interval_in_explicit_plan(schedule)) {
        if (interval_is_assigned_to_entity(member)) {
            remove_interval_from_explicit_plan();
            deallocate_interval();
        }
    }
    if (explicit_plan_is_empty(schedule)) {
        wake_up_processes_waiting_for_plan_to_end();
    }
    return 0;
}
```

4.1.2 Applications

The explicit plan SDF is flexible enough to be used in several different applications. However, the application structure that was used as a driving example for the SDF is shown in Program 4.5. This design philosophy, in which the application semantics are used to create the scheduler, is fundamentally different from the conventional method, in which the scheduler semantics are used to design an application.

An application utilizing the explicit plan SDF first creates a group that is associated with the SDF. If there are other groups already present in the system, then the application needs to place this group in the scheduling hierarchy by joining at least one of the existing groups. The parent thread of the application then forks several child processes. Each of its children simply enters a loop in which it repeatedly executes a computation and then suspends itself until its next allocated interval. The parent thread joins each child to the explicit plan group and constructs a schedule for the child threads. After the schedule is created and all the child threads have been created, the parent submits the schedule to the explicit plan SDF and waits for it to finish. Each child thread is then allowed to execute only during its assigned interval. When the schedule finishes, the parent resumes execution. It removes each child thread from the explicit plan group and destroys it before exiting.

Program 4.5 An Application Utilizing the Explicit Plan Programming Model

```
void main(void) {
    open(group_scheduling_device);
    create_group(EXPLICIT_GROUP, EXPLICIT_SCHEDULER);
    for (i = 0; i < num_threads; i++) {
        pid[i] = fork();
        if (pid[i]) {
           /* Parent */
           join_group(pid[i]);
           add_process_to_explicit_plan(pid[i]);
        } else {
            /* Child */
            while (!suspend()) {
                computation();
            }
            exit();
        }
    }
    submit_explicit_plan_and_wait();
    for (i = 0; i < num_threads; i++) {
        leave_group(pid[i]);
    }
    destroy_group(EXPLICIT_GROUP);
 }
```

4.2 Pipeline Model

Many real-time applications require multiple tasks to coordinate their execution, and a common model for this cooperation is the pipeline model. Applications such as video-conferencing programs[7] and industrial automation control can be represented as a pipeline of several tasks. The pipeline model is illustrated in Figure 4.2.

Under other operating systems, the pipeline model can be implemented using a static priority scheduler. However, the the unified scheduling model allows for application-specific scheduling. For example, if a stimulus interrupt occurs during the processing of a previous stimulus, then the interrupt service routine can be scheduled to execute after the current iteration of the pipeline finishes. Furthermore, if one of the computations blocks, then the semantics of the pipeline may allow other parts of the pipeline to either wait for the computation to unblock or to execute concurrently.

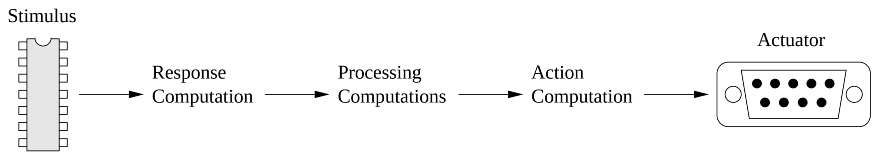

Figure 4.2: Pipeline Application Model

4.3 E-Machine

The E-machine is an embedded interpreter that is designed for real-time applications[5]. It divides computations into two classes: drivers and tasks. Drivers are computations which interact with hardware and are assumed to take zero logical time. Tasks perform processing on the data and are assumed to take a positive amount of logical time.

The E-machine consists of a single thread that is responsible for executing E-code. E-code can execute a driver, schedule a task, and schedule the execution of E-code in the future.

This programming model can be directly implemented by the unified scheduling model. The E-machine thread must be scheduled to execute during specific time intervals. Therefore, these semantics can be realized through the explicit plan scheduler. The task threads are scheduled using an earliest-deadline-first algorithm, and their execution yields to the E-machine thread. Thus, a static priority scheduler, which expresses the precedence of the E-machine thread over the tasks, can be used at the top level. Groups associated with the explicit plan SDF and an earliest-deadline-first SDF can be used to schedule the E-machine and tasks accordingly. The structure of such a hierarchy is shown in Figure 4.3.

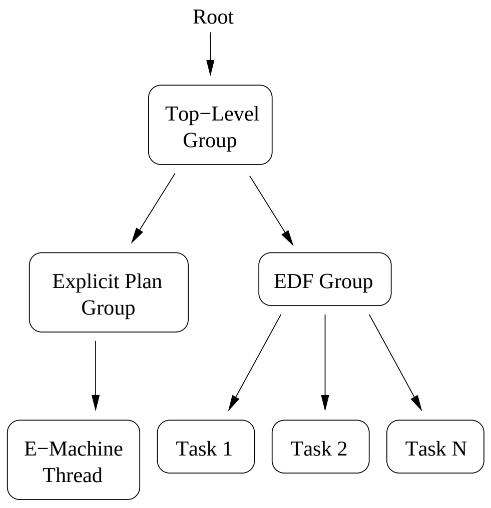

Figure 4.3: E-Machine Scheduling Hierarchy

4.4 Scheduling Overhead

The overhead of executing the scheduling hierarchy depends on the structure of the hierarchy and the decision path that it takes. A distribution of the execution time taken by the hardirq and softirq hierarchy shown in Figure 3.5 is given in Figure 4.4. This test was performed over a period of 10,000 seconds on a computer system with a 200 MHz Pentium Pro processor, generating approximately 42 million data points. System load was generated by cyclically executing a two-process kernel compilation on the local hard disk through a secure shell connection. The data shows that in most cases the scheduling hierarchy takes little time to make its decision. For reference, a distribution of the standard Linux 2.4.22 process scheduling algorithm overhead is shown in Figure 4.5. This data was gathered under identical test conditions and generated roughly 1 million data points.

4.5 Interrupt Accuracy

The modifications to interrupt concurrency control that have been made allow for interrupts to be delivered to the CPU more accurately. This fact has been confirmed by

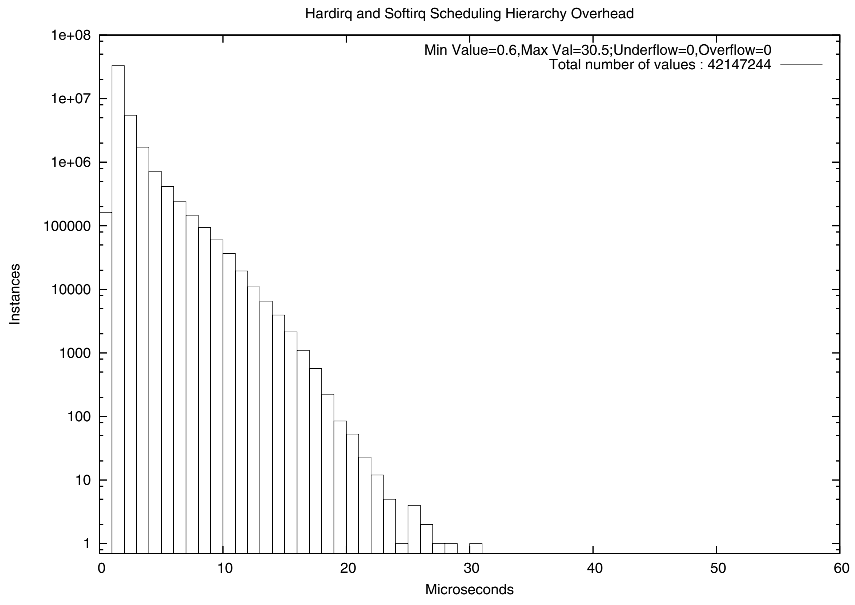

Figure 4.4: Hardirq and Softirq Scheduling Hierarchy Overhead

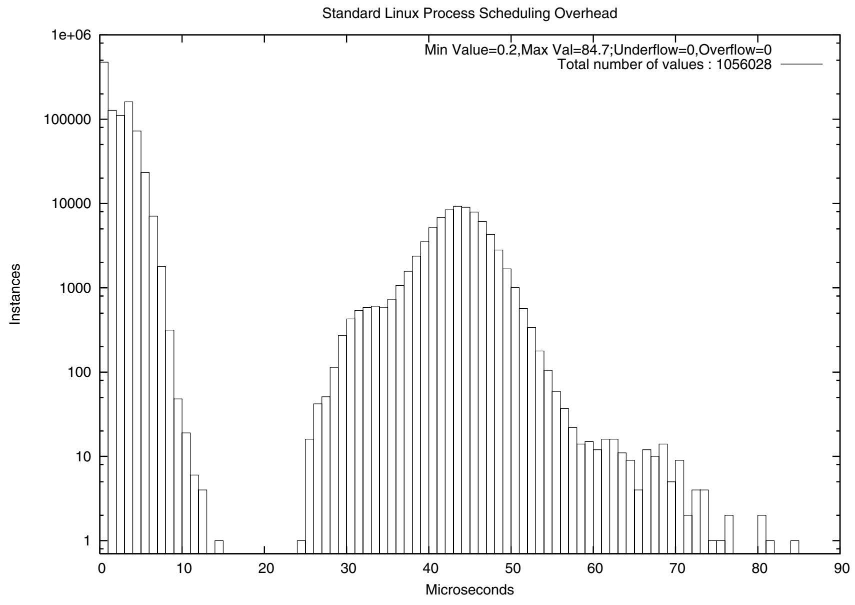

Figure 4.5: Standard Linux Process Scheduling Overhead

measuring the accuracy with which a timer interrupt can be delivered to the CPU. The difference between the scheduled time of a timer interrupt and its actual delivery in a system with interrupt modifications disabled is shown in Figure 4.6. This can be compared to the same measurement under a system with interrupt modifications enabled, which is shown in Figure 4.7. The system with interrupt modifications increases accuracy and prevents the outliers that are present in the system without interrupt modifications. These tests were performed over a period of 10,000 seconds on a computer system with a 200 MHz Pentium Pro processor, and both generated approximately 1 million data points. System load was generated by cyclically executing a two-process kernel compilation on the local hard disk through a secure shell connection.

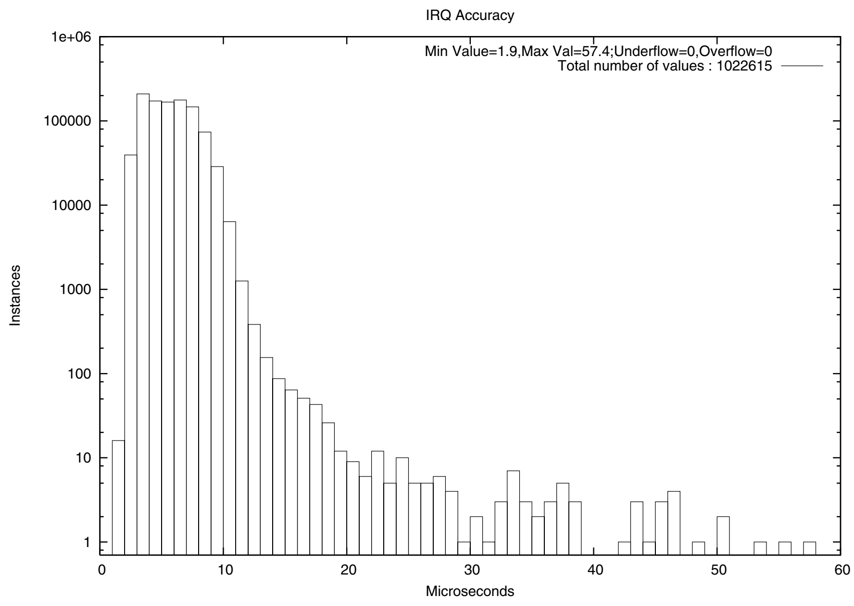

Figure 4.6: Interrupt Accuracy Without Interrupt Modifications

4.6 Summary

The explicit plan scheduler illustrates the ability of the unified scheduling model to express the semantics of an existing programming model. Furthermore, it shows how the flexibility that is provided by the unified scheduling model can be exploited in

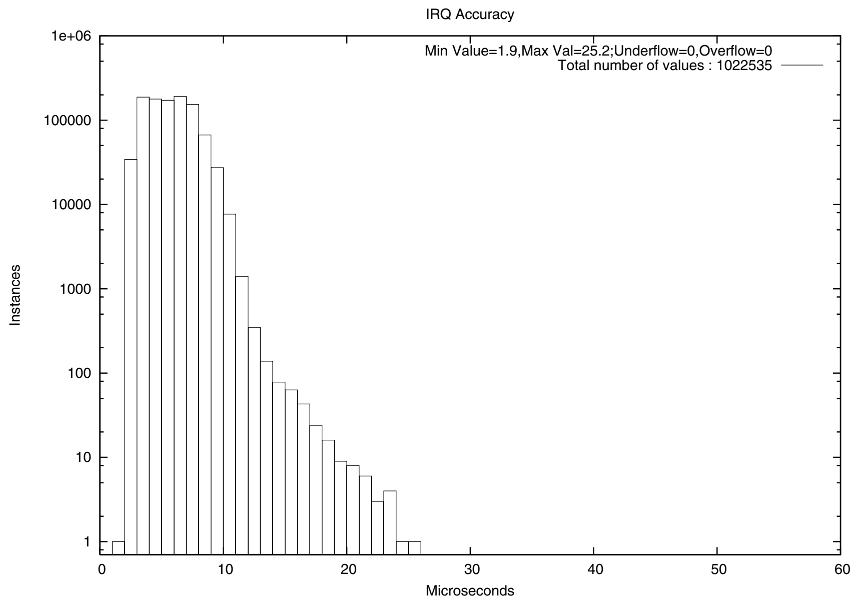

Figure 4.7: Interrupt Accuracy With Interrupt Modifications

order to improve a given programming model. The pipeline model and the E-machine are further examples of how the unified scheduling model can allow a system designer to accurately implement a desired programming model.

The scheduling overhead results show that if the scheduling hierarchy is structured correctly, it can frequently make its decision with little overhead per invocation. The interrupt accuracy data demonstrates how the modifications made to interrupt concurrency control allow for greater computational accuracy than is otherwise possible.

Chapter 5

Conclusions and Future Work

The demand on computer systems to handle an increasingly wide range of applications with equally varied execution semantics creates a need for a fully configurable resource control mechanism. Since control of system resources can be simplified to controlling all the computations that use them, a configurable computation scheduling system is needed. Furthermore, the system must be able to unite all computations under a single scheduling mechanism in order to attain the precision that is desired by system designers.

5.1 Proxy Execution

Although the precision with which computations can be controlled has been increased significantly, there is room for even more improvement. The concurrency constraints related to process preemption can be refined in order to allow proxy execution.

Linux does not allow preemption of a process while it is holding a lock. If this restriction was not enforced, then deadlock would occur when a process was preempted while holding a particular lock needed by another process that subsequently attempted to acquire the same lock. This limitation is not beneficial, especially for processes with real-time constraints, because it increases process scheduling latency. Just as the addition of SMP concurrency control facilitated the creation of the preemption patch, the addition of a unified scheduling model allows for the creation of even finer concurrency and preemption control.

Proxy execution is a form of optimistic concurrency control, as it allows for conflicts to occur. Deadlocks are avoided by detecting conflicts and fixing them. Proxy execution involves tracking which computations hold particular locks. When this feature has been added, full preemption can be implemented to allow any computation to run at any moment, regardless of lock state. When a lock conflict arises, the unified scheduling model can run the thread which holds a needed lock only until it relinquishes the lock. This allows the lock-holding computation to run only as a proxy for the computation whose execution is desired.

5.2 Computation Type Reduction

A number of computation types have been created throughout the evolution of operating systems. Entities such as bottom-halves, softirqs, and tasklets were each developed in response to the demand from system designers to have the ability to specify new computational constraints.

Since the creation of computational types has been driven by the need to express scheduling semantics rather than inherent differences in their nature, many of these computational types are no longer necessary under the unified scheduling model. System designers can now explicitly specify the circumstances under which each computation is executed. Therefore, computation types only need to be established when there is a fundamental difference in their execution method.

Currently, there are only two types of computations that fundamentally differ in how they are executed, those that must run in their own context (context-owning computations) and those that can be run in any context (context-borrowing computations). Each existing computation type could be placed in either of these categories. Their scheduling semantics could then be expressed by their placement in the scheduling hierarchy rather than their computation type.

5.3 Control Path Consolidation

The Linux operating system contains numerous control paths. This is due to the fact that various decisions are made outside of a centralized scheduling nexus, and thus create forks in a given execution path. The large number of control paths hinders evaluation of code correctness, especially when analyzing concurrency control.

Because all computations have been unified under a single scheduling mechanism, the flow of control within the operating system can be significantly simplified. Figure 5.1 illustrates an example of a standard Linux code path alongside the ideal code path under the unified scheduling model. The ideal code path is as follows:

1. When a system event occurs, update the appropriate data structures to indicate new information which might affect the scheduling decision.
2. Give the scheduling hierarchy an opportunity to make a choice based on the new information.
3. Begin execution of the computation that the scheduling hierarchy chooses.
4. Continue executing the computation until it either finishes or the CPU receives an interrupt.
5. Repeat this sequence indefinitely.

Large portions of the standard Linux code paths still exist. They have simply been modified to utilize the unified scheduling model. Since the scheduling hierarchy only needs to be called from a single place in the operating system, these code paths can be consolidated.

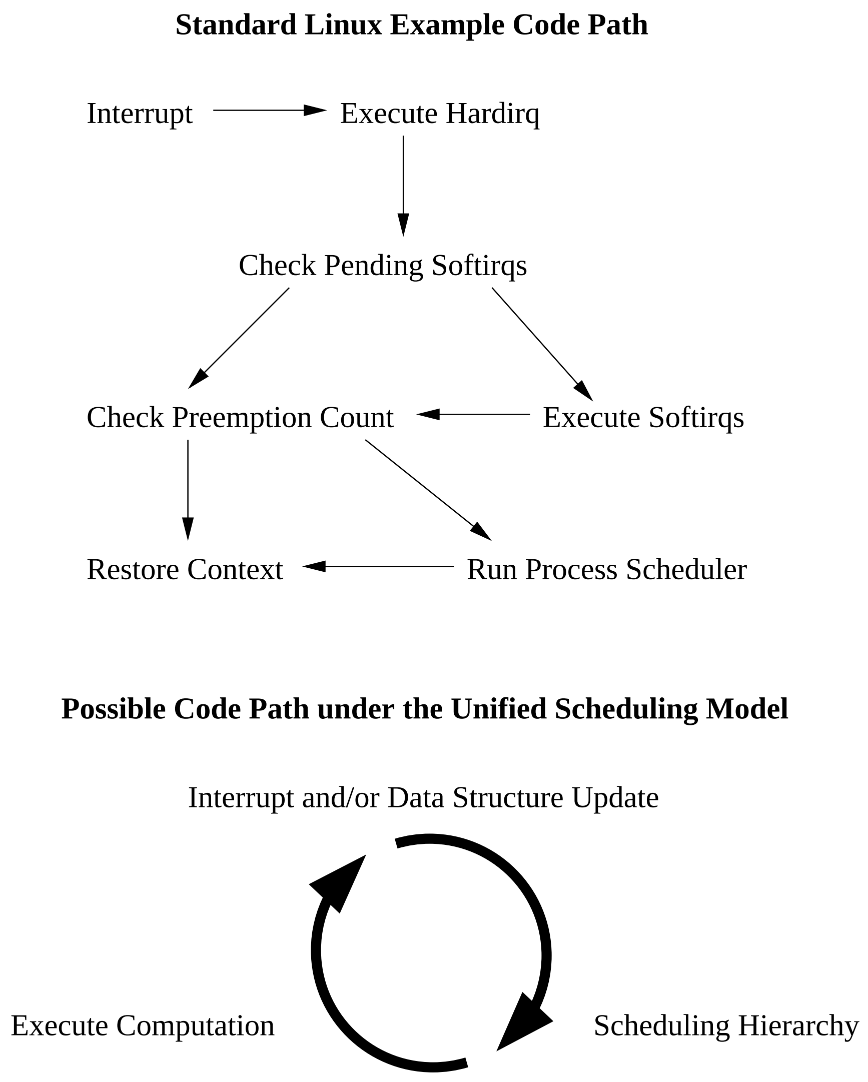

Figure 5.1: Code Path Comparison

Bibliography

[1] G. Bollella and K. Jeffay. Supporting Co-Resident Operating Systems. In *Proc. Real-Time Technology and Applications Symposium*, pages 4–14, June 1995.

[2] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, Inc., 2nd edition, 2003.

[3] L. Torvalds et al. The Linux Kernel Archives. http://www.kernel.org.

[4] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., 1995.

[5] Thomas A. Henzinger and Christoph M. Kirsch. The Embedded Machine: Predictable, Portable Real-Time Code.

[6] R. Hill, B. Srinivasan, S. Pather, and D. Niehaus. Temporal Resolution and Real-Time Extensions to Linux. Technical Report ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, University of Kansas, June 1998.

[7] K. Jeffay. On Latency Management in Time-Shared Operating Systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 86–90, May 1994.

[8] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[9] 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, 1973.

[10] LynuxWorks. http://www.lynuxworks.com.

[11] K. Ramamritham and J. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. In *Proceedings of the IEEE, vol. 82, no. 1,* pages 55–67, January 1994.

[12] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems.* PhD thesis, University of Virginia, May 2001.

[13] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving Real-Time Systems Using Hierarchical Scheduling and Concurency Analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 25–36, Cancun, Mexico, December 2003.

[14] J. Regehr and J. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, December 2001.

[15] RTAI. http://www.rtai.org.

[16] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* John Wiley & Sons, Inc., 6th edition, 2002.

[17] B. Srinivasan. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. Master's thesis, University of Kansas, February 1998.

[18] Wind River Systems. VxWorks Programmer's Guide.

[19] TimeSys. http://www.timesys.com.

[20] V. Yodaiken. The RTLinux Manifesto. In *Proc. of The 5th Linux Expo, Raleigh, NC*, March 1999.