A Flexible Scheduling Framework Supporting Multiple Programming Models with Arbitrary Semantics in Linux*

Noah Watkins
Systems Lab
UC Santa Cruz
jayhawk@soe.ucsc.edu

Jared Straub and Douglas Niehaus
Information and Telecommunication Technology Center
University of Kansas, Lawrence, Kansas 66045
iqrally@ittc.ku.edu, niehaus@ittc.ku.edu

Abstract

We present a hierarchic scheduling framework for Linux called Group Scheduling that facilitates the creation of arbitrary thread schedulers. Traditional approaches to developing new scheduling semantics require semantic mappings onto existing schedulers, such as static-priority. Group Scheduling allows for a direct implementation of semantics, allowing clear mappings at any level a developer desires. In order to effectively support scheduling semantics, integration with concurrency control is necessary (e.g. priority inheritance). However, when considering arbitrary scheduling semantics hard-wired solutions such as PI can't adapt. We present Proxy Execution as a general mechanism to resolving policy conflicts that arise as tasks from different scheduling domains interact through the RT-Mutex primitive.

1 Introduction

The recent and continuing evolution of computer systems and their applications exhibits a significant semantic explosion. The capabilities of computer systems continue to grow rapidly at the hardware level, and the range of application semantics is keeping pace. However, where hardware enhancements of the past have centered around increased chip frequencies, today we are seeing an increased degree of parallelism present in single systems. Effectively exploiting this parallelism in many instances requires that a system support diverse application semantics and reconfigurability.

Semantic diversity and reconfigurability of a single system means that the system supports multiple application semantics, that the set of semantics used by applications executing on a system can be reconfigured, and that applications with different semantics can coexist. For example a single system should be capable of supporting relatively familiar application semantics such as deadline, rate, priority, and CPU share based semantics, in addition to experimental and emerging semantics that may include application progress or other semantics that do not map well onto traditional scheduling policies. Moreover, the set of applications on a system cannot only require a wide variety of execution semantics through reconfigurability, but in fact may require *many* or *all* of those semantics simultaneously. Single system support of multiple semantics can also reduce hardware costs by replacing several separate systems with a single physical platform that accurately supports the semantics of all applications.

In comparison to the huge amount of change in system hardware support, the semantics of the

*The work presented here was supported in part by NSF grants CNS-0716740 and CCF-0615035

programming models provided by modern operating systems has scarcely changed in the last 30 years. Threads are still typically scheduled with dynamic priority semantics [1], and concurrency is most commonly controlled using semaphores. Some expansion of semantics in scheduling and concurrency control has, of course, occurred in the past three decades. In particular, condition variables, reader-writer locks (including RCU), queuing semaphores, and monitors have expanded the concurrency control domain, while rate-monotonic analysis and EDF have expanded the availability of scheduling semantics in systems with static semantic configurations [1,2]. Nonetheless, it is fair to say that a software developer transported from 1980 would have far less difficulty recognizing and understanding the semantics of scheduling and concurrency control offered by modern operating systems than they would the hardware supporting today's mix of operating systems and applications.

1.1 The Growing Semantic Gap

The semantic gap between applications and operating systems is growing for two major reasons. First, the diversity of applications is expanding as cheap, powerful hardware begs to be used in new, imaginative ways. The second force fueling this semantic divide is the tremendous inertia the dynamic priority programming model has in modern operating systems due to widespread deployment, maturity, and simplicity. Decades of scheduling research and real-time system development have been done in environments implementing priority models, and this model is taught to all students of computer science.

While we do argue that a tremendous problem is arising with an increasingly large semantic gap, we in no way argue against the importance of priority-based programming models. On the contrary, priority programming models are by far the dominant models, and in an important sense a *victim of the their own success*. Many applications exhibit priority semantics directly, and many other application semantics are easily mapped onto the priority model. Concurrency control mechanisms developed assuming priority-based scheduling are convenient and easier to develop. Finally, and perhaps most importantly, developers of new applications with new semantics often have no alternative to mapping their semantics onto priority. This lack of choice may be a result of having no access to OS source, or the prospect of developing a new scheduler may pose an insurmountable barrier to developers because of time constraints, or lack of expertise. Bold developers sometimes use complex middleware solutions mapping application semantics onto traditional static-priority programming models. However, as application semantics become more sophisticated and diverse, the required mappings become more difficult and costly to create.

1.2 Bridging the Gap

A common pattern has emerged due to the great difficulty of changing the semantics of programming models exported by an operating system. Practitioners commonly assume that they will have to use one or more of the following techniques: (1) adaptation of their applications to use priority semantics, (2) manipulation of priorities to force the desired semantics at the user-level, or (3) using concurrency control mechanisms for their scheduling effects. These techniques have often proved to be *good enough* on dedicated systems where reasoning about a single semantics, or several simple semantics is a tractable problem. However, the explosive growth in application semantics, and the combination of semantics coexisting on a single system, is making such approaches increasingly difficult. Even worse, researchers are increasingly interested in programming models that make it *easier* to formally model and verify application behavior. Continuing to bridge the semantic gap by mixing and matching indirect methods will not scale as the resulting complexity will be too great. An alternative approach is to directly implement application semantics, allowing developers to choose the appropriate level at which semantic mappings occur.

Methods for *directly* implementing a wide range of application semantics, and for specifying how the conflicting demands of applications with differing semantics can be reconciled, would greatly simplify much of the *accidental* complexity incurred by techniques such as indirect semantic mappings. While direct implementations do not reduce the *inherent* complexity of application semantics, a direct implementation significantly reduces the complexity *added by indirect methods*, and facilitates modelability.

1.3 Group Scheduling

Group Scheduling [2](GS) is a practical approach to hierarchic scheduling which emphasizes *direct repre-*

[1] The introduction of the CFS scheduling class as a replacement for the SCHED_OTHER implementation is a step in the right direction, away from multiplexing many semantics into a priority-based scheduler

sentation of computation structure and *direct implementation* of application semantics [6]. For many years Group Scheduling has been used for a wide variety of projects, proving capable of describing a broad set of application semantics, and expressing system level policies such as how to balance conflicting demands from different applications. The use of GS reduces application implementation complexity compared to using indirect methods because developers are free to express semantics in ways appropriate for a given application.

This paper describes the most recent extension of Group Scheduling, which has integrated scheduling with the semantics of the RT-Mutex concurrency control primitive. This work builds upon, and significantly generalizes, the concurrency control approach used in CONFIG_PREEMPT_RT [4].

Our approach, which we call Proxy Execution, employs a general representation of mutex blocking relations, and provides hooks into which customized routines may be plugged to make decisions such as granting mutex ownership on release, and permission to steal [3]. The goal of Proxy Execution is to schedule owners of mutexes in such a way that desirable tasks blocked on mutexes are made runnable as soon as possible. A common solution, and the one used in CONFIG_PREEMPT_RT, is to implement the priority inheritance protocol. However, unlike PI, Proxy Execution works independently of scheduling semantics, and resolves policy conflicts that cross scheduling domains. While Proxy Execution is a simple idea, the implementation is complicated by a number of factors. These include the maintenance of blocking graphs, memory allocation requirements, and limitations on concurrency incurred during maintenance of the graph.

Proxy Execution does not yet integrate all forms of concurrency control, but we believe that the approach we have taken to the integration of scheduling with concurrency control already provides a platform within which a wide range of programming model semantics can be directly implemented, including a large number of scheduling algorithms popular in the research community that are currently implemented in specialized kernels, or in Linux using *ad-hoc* methods and evaluated with micro-benchmarks.

The availability of a common platform within which a wide range of algorithms and competing semantics could be implemented would be a significant advantage in facilitating accurate comparisons. Since each algorithm could be tested using identical system configurations, system performance measurements can be taken to accurately reflect the characteristics of a particular algorithm or scheduling policy.

Many methods exist for collecting performance related information on Linux-based systems, both in user-space and within the kernel. Common frameworks include FTrace, LTTng, SystemTap, and Kprobes. In this paper we make reference to one such framework called DataStreams, developed and used by the KUSP research group.

The remainder of this paper is organized as follows. First we briefly explore related work. Next we provide a background and motivation for the development of Proxy Execution, and describe its implementation in Linux. Finally we provide a sampling of programming models implemented in Group Scheduling to illustrate its power and generality.

2 Related Work

The original Group Scheduling [5,6] framework was built in the 2.4 Linux kernel, and represented all computation types such as hard-IRQs, soft-IRQs, and tasklets, as members in the hierarchy. However, in this version integration of scheduling semantics with concurrency control was not addressed.

The work of [3] adapted the Group Scheduling framework to function on top of CONFIG_PREEMPT_RT, which in turn unified all computations as threads, allowing a simplification of the Group Scheduling framework's treatment of computation types. This work also introduced integrated concurrency control with a limited form of Proxy Execution that required an SDF to consider a thread blocked on a mutex. Knowledge of blocking relationships then allowed the framework to calculate the correct task to run to resolve the conflict. This form of Proxy Execution did not support scheduling algorithm optimizations such as removing a blocked task from its run-queue, nor did it address support for SMP systems.

The scheduling stack introduced at the same time as the Completely Fair Scheduler represents a first attempt at supporting multiple scheduling domains in Linux. However, the strict precedence order of scheduling domains and hard-coded assumptions of a priority scheduling model in the RT-Mutex framework are not flexible enough to support arbitrary semantics.

[2] Group Scheduling in this paper does not refer to the kernel facility *cgroups*, once named Group Scheduling

[3] stealing

Priority inheritance and priority ceiling are classic approaches to solving the problem of priority inversion [7]. The RT-Mutex in Linux implements the priority inheritance protocol [4], however neither solution effectively supports a general scheduling policy, and forces developers to map all policies onto priority, further complicating the situation when multiple policies are simultaneously active on a system.

3 Implementation

We describe our implementation of Proxy Execution, including the extensions to the RT-Mutex framework, and the integration of these extensions with Group Scheduling to support Proxy Execution. We begin with a brief overview of the Group Scheduling framework, and provide a detailed motivation for Proxy Execution. Then we describe Proxy Management, which is our set of extensions made to the RT-Mutex primitive that support Proxy Execution. Finally, we provide a detailed view of Proxy Execution and it's use within the Group Scheduling framework.

3.1 Background

Group Scheduling is a hierarchic scheduling framework consisting of the following components: (1) scheduling decision functions (SDFs), (2) groups of computations, and (3) scheduling data. An SDF is an implementation of a specific scheduling semantics (e.g. static-priority or EDF), and controls the computations of a group, that is, a group represents a set of computations scheduled according to the scheduling semantics implemented by the controlling SDF. A computation is represented as a member of a group, and may be a thread or another group (allowing a hierarchic structure). And finally, arbitrary scheduling data may be associated with both groups, and members of groups.

Scheduling Decision Functions

The idea of an SDF is analogous to a *scheduling class* in the Linux kernel. Like a *scheduling class* an SDF is implemented by filling in generic function pointers. In fact, Group Scheduling has reused many of the scheduling hooks used in the implementation of the existing Linux scheduling framework.

An SDF itself is only an implementation of a scheduling semantics, and like a *scheduling class* which operates on a run-queue of tasks in a scheduling domain, an SDF too must operate with a set of computations. Group Scheduling organizes computations into groups, where a computation may be a thread, or another group. A group must also be associated with exactly one scheduling decision function, which controls the semantics by which the group members are scheduled.

GS Hierarchy Evaluation

As discussed, groups are composed of computations, which may include other groups. This allows a set of groups to be organized in a hierarchic structure, with a single root group. When a scheduling decision is made the hierarchy is traversed starting with the root group. As the hierarchy is traversed, each group evaluates its associated scheduling decision function over its members, returning a group, a thread, or a control message such as *no decision*. This organization is very general, allowing multiple scheduling semantics to be active on a system simultaneously, but poses difficult problems for resolving conflicts between tasks as they interact through shared resources. In this paper we consider a shared resource to be represented by a semaphore, and a conflict to refer to a blocking relationship between a waiter and an owner of a semaphore.

An example hierarchy is shown in Figure 1. The root group is labeled *SEQ*, short for *sequential SDF*, and implements a semantics that chooses the first runnable task in a queue. When this hierarchy is evaluated each member is recursively evaluated until a runnable thread is found. Threads are depicted by the squares labeled $T1$ through $T6$. The group labeled *Linux* represents a scheduling decision made by Linux through the evaluation of the standard Linux scheduling classes.

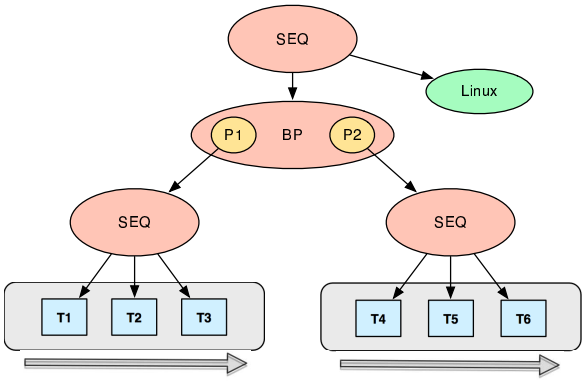

FIGURE 1: *Balanced Progress Hierarchy*

Resolving Policy Conflicts

Group Scheduling allows for a completely general organization of scheduling semantics to exist on a system. However, consider one common type of conflict that results from such a flexible configuration. Suppose a task scheduled under a static priority policy blocks on a mutex owned by a task scheduled under CPU reservation policy. Resolving this conflict in the general case is very difficult as no general decision function can be created to compare arbitrary policies. The solution that Group Scheduling provides is called Proxy Execution. Proxy Execution is a general mechanism for resolving such conflicts. However, first let us examine how Linux resolves conflicts between distinct scheduling domains.

Linux implements two main *scheduling classes*, the real-time static priority scheduler, and the completely fair scheduler. These two classes are organized in a stack with a strict ordering of importance. Specifically, the real-time class is always considered first, followed by the CFS class, and finally the class which always chooses the idle thread. In order for Linux to effectively support the overall, system scheduling policy implied by the strict ordering of scheduling domains a mechanism must be in place to resolve the conflict that result when the execution of a thread from one scheduling domain is blocked by a thread from another domain, the most important case in Linux being a real-time task blocking on a mutex owned by a task scheduled under the CFS policy.

The solution used in Linux is to implement the priority inheritance protocol, thereby integrating scheduling semantics with concurrency control. However, the priority inheritance protocol is only applicable to tasks scheduling under priority semantics, thus the PI mechanism alone cannot resolve such a policy conflict. The solution used is two-fold: (1) all tasks are capable of being moved into the real-time scheduling class (i.e. a priority field exists in the task struct), and (2) a strict separation of priority exists between threads in the real-time domain, and those in the CFS domain. Given these properties the clever solution is to simply observe alterations to a task's priority value and automatically move a task from the CFS domain to the real-time domain, and vice versa, depending on which range a priority falls into, either the real-time class or the CFS class. This mechanism utilizes the effects of the PI protocol to achieve a more complex system policy. In the general case this problem is very difficult to solve, and any solution is likely to fail if it builds on top of existing forms of integration that assume specific system policies, such as priority.

One very important aspect of the solution implemented in Linux is worth considering, notably the *movement* of a task from one domain to another as a component of the solution to resolve policy conflicts between domains. In fact, Proxy Execution uses a similar method, but generalizes the implementation and addresses some short-comings with the existing Linux mechanism. Specifically, the movement of a task from one domain to another implies that each task be *capable* of moving, that is, have the proper data structures. However, in Group Scheduling arbitrary and dynamically allocated scheduling parameters can be associated with computations, thus the overhead of supporting a computation's direct movement between any task would require modification to all member computations as a result of updating a single group's configuration. Second, Group Scheduling places no restriction on the number of group memberships a computation may have. That is, a thread may belong to multiple scheduling domains. Obviously, the concerns are similar to that of direct task movement, but more importantly, the order of hierarchy traversal is non-deterministic, thus all memberships must be persistent.

3.2 Integrated Concurrency Control

Linux integrates priority scheduling semantics with concurrency control by hard-wiring an implementation of the priority inheritance protocol into the RT-Mutex primitive. Through priority boosting CONFIG_PREEMPT_RT implements a limited form of proxy execution, while scheduling semantics are further integrated into the RT-Mutex implementation during lock release and stealing operations by using priority specific decision functions. These two components, proxy execution and semantic integration, form what we refer to as a complete integration of scheduling semantics with concurrency control. A general complete integration thus requires a general treatment of both components.

The integration of scheduling semantics with concurrency control is illustrated by the priority specific decision functions used in the RT-Mutex implementation. During lock release the highest priority waiter is chosen to become the pending owner, and the priority of a task is again compared to a pending owner to determine if a lock can be stolen. The former operation involves a function evaluated over all waiters on a mutex, and the later is a comparison made between two tasks. Group Scheduling generalizes these decisions using hooks in the RT-Mutex implementation that can be filled in with

Group Scheduling functions that directly implement specific scheduling semantics.

A general treatment of proxy execution requires a general representation of the blocking relations that exist between tasks interacting through a mutex. In PREEMPT_RT these relations are implicitly represented through priority boosting. The nature of the priority inheritance protocol allows for a simple implementation that accumulates the maximum waiter priority at each mutex, and passes the maximum priority value onto the owner. This mechanism is attractive because of its constant memory requirements and simple internal concurrency control, but it masks individual blocking relations making it difficult or impossible to implement semantics with more complex requirements, such as CPU bandwidth limiting, which requires explicit links between a blocked task and its proxy in order to correctly perform resource accounting.

Group Scheduling uses a facility called Proxy Management to create and manage explicit representations of blocking relationships. These relationships are then used by the Group Scheduling framework to implement Proxy Execution.

3.3 Proxy Management

Proxy Management refers to the set of Group Scheduling extensions built into the RT-Mutex framework that track blocking relations between tasks. The difference between Proxy Management and Proxy Execution is that Proxy Management refers to the representation, construction, and maintenance of task-to-task blocking relations, while Proxy Execution refers to the use of this information in Group Scheduling to resolve scheduling policy conflicts.

There are two types of blocking relations tracked by Proxy Management. The first is the relation that describes a task being blocked, either directly or indirectly, on another task holding a resource. The second type of relation maintained by Proxy Management is the *proxy relation*. This relation is also a blocking relation, but represents the first conflict that must be resolved in order to "unblock" a task as soon as possible. While a task may be a part of any number of blocking relations, it is always associated with exactly one proxy relation. Intuitively a proxy relation tracks the head of a given locking chain, however in practice determining a proxy is an iterative process that involves walking the lock chain.

The implementation of Proxy Management uses two distinct data structures, one to represent generic blocking relations, and a second to represent proxy relations. Additionally, changes to the task_struct and existing RT-Mutex data structures were necessary.

3.3.1 Blocking Graph Representation

Conceptually a blocking graph is a set of blocking relations linked together to reflect the state of a locking chain. Blocking graphs in Proxy Management explicitly represent all blocking relations in a chain, both direct and indirect, by linking together data structures representing single blocking relations. The data structure used to represent a blocking relation is the *Waiter Node*. This structure represents a blocking relation between two tasks, T_1 and T_2, and a lock L_1, where T_1 is the owner of L_1 and T_2 is blocked, either directly or indirectly on L_1. Consider the graph in Figure 2. This figure depicts two indirect, and four direct, blocking relations represented by the nodes labeled $N_{2,3,4,5}$.

A *Proxy Relation* refers to the blocking relation that exists between a blocked task, and the owner of the mutex at the head of the blocked task's locking chain. This type of relation is represented by the *Proxy Waiter* structure. For example in Figure 2 four proxy relations exist between tasks $T_{2,3,4,5}$ and T_1, the owner of the lock L_1 at the head of the chain. The *proxy relation* data structures in Figure 2 are labeled $W_{2,3,4,5}$. Each of the dashed lines in the figure represent linked lists that are used to organize the Proxy Management data structures, and their association to each other and to the existing components of the RT-Mutex framework.

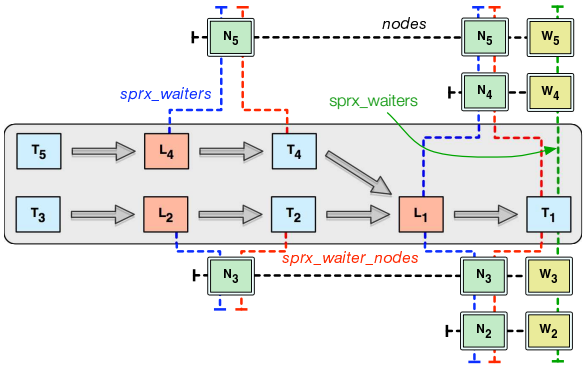

FIGURE 2: *Proxy Management data structure representation*

3.4 Proxy Execution

Proxy Execution is the Group Scheduling mechanism used to resolve arbitrary scheduling conflicts that arise when a task blocks on a mutex. Consider the state of the system shown in Figure 2 in which tasks $T_{2,3,4,5}$ are blocked on task T_1. A Group Scheduling policy that selects one of these blocked tasks as the most desirable task will use Proxy Execution to resolve this conflict by instead running T_1 as a *proxy* of the desired blocked task. Notice the correspondence between task T_1 and the Proxy Relation associated with each blocked task. In fact, Group Scheduling implements Proxy Execution using the proxy relations created by the Proxy Management framework.

Proxy relations alone cannot be used to implement Proxy Execution because a proxy relation represents a scheduling policy agnostic view of a blocking relation. That is, it reflects only the task-level view of a relation. This is in conflict with the goals of Proxy Execution which by definition resolve conflicts between specific scheduling policies. Additionally, two tasks on a system can be involved in at most a single blocking relation with each other. However, Group Scheduling allows tasks to hold memberships in multiple groups. Thus, a proxy relation at the task-level corresponds to proxy relations between one or more Group Scheduling members in possibly distinct scheduling domains. Group Scheduling solves these problems through the use of *avatars*.

An avatar is a special-purpose member scheduled in-place of a specific member corresponding to a task blocked on a mutex. An avatar is created for each membership of a task associated with a Proxy Relation, and may use scheduling parameters identical to those of the task the avatar masquerades as. Proxy Execution is thus achieved by scheduling the blocking task associated with the Proxy Relation that prompted the creation of the avatar. Intuitively the use of an avatar is similar to giving a task temporary group membership under a set of different scheduling parameters. In practical terms an avatar is automatically allocated for each member because a task can only ever be associated with at most one proxy relation. In contrast, it is possible that a task may need to be scheduled as a proxy, simultaneously, from all scheduling domains.

3.4.1 Scheduling Hooks

Avatars are brought in and out of existence as proxy relations are created and destroyed. When a new avatar is created the appropriate SDF is notified by calling the *insert-member* hook, and specifying the avatar as the member being added to the group. Likewise the generic *remove-member* hook is called to remove an avatar when a proxy relation is destroyed by the Proxy Management framework. An SDF may examine a member to determine if it is a proxy allowing it to execute any special setup or tear-down routines that are required.

3.5 RT-Mutex Extensions

This section covers the extensions made to the RT-Mutex framework that implement Proxy Management. In this section we refer to proxy relations that are created and destroyed. These terms convey heavy-weight operations, however in many circumstances the implementation is improved by re-using data structures.

3.5.1 Blocking

Blocking on a mutex is an operation that results in the creation of at least one new proxy relation. When a task blocks on a mutex a direct blocking relation is always created between blocking task and the owner of the mutex. In addition to the direct blocking relation, zero or more indirect relations are created if the blocking task has existing waiters, and when the owner of the lock being blocked on, is also blocked on a mutex. In the later case the locking chain is "walked" by iteratively examining blocking relations until an owner is found that is not blocked on a mutex. In both cases existing proxy relations are destroyed and new relations created as locking chains are extended.

For each proxy relation that is created or destroyed the scheduling framework is notified by inserting or removing an avatar associated with the affected proxy relation.

3.5.2 Releasing

The release of a mutex must be integrated with scheduling in the following ways: (1) the selection of a pending owner from among the set of direct waiters is integrated with scheduling, and (2) proxy relations must be updated to reflect the change in ownership of the mutex. First, the selection of a pending owner is implemented as a hook into the Group Scheduling framework that allows the decision function to be dynamically chosen. Second, the Group Scheduling framework must be notified of changes made to proxy relations.

Releasing a mutex is a two-step process that updates the proxy relations for all threads waiting on the mutex to be released. In the first step the proxy relations that exist between all waiters on the mutex and the owner are destroyed, including the relation involving the newly selected pending owner. Second, all of the relations except the one involving the pending owner are re-created between the original waiter associated with the relation, and the pending owner. It is important to note that proxy relations are created for a pending owner during release, as opposed to when actual ownership takes place. Even though the choice of pending owner is integrated with scheduling, no guarantee is made that this choice will *actually* be scheduled immediately. Thus, proxy relations must be created to prevent the situation in which a pending owner is never scheduled to fully acquire a mutex.

Stealing

The ability to steal a mutex is an optimization created to exploit the window of time between a pending owner being selected, and it running to fully acquire the lock. The stealing operation is integrated with scheduling in a nearly identical way to that of releasing a mutex. First, the decision to steal a lock from a pending owner must be integrated with Group Scheduling. This decision is implemented as hook that can be replaced by Group Scheduling. Its semantics are dependent on the Group Scheduling policy and configuration, and must be able to make a boolean valued comparison between tasks of potentially different scheduling domains.

When a task attempting to steal a lock is denied permission to take possession of the lock it immediately blocks on the mutex. If a task does steal a lock then it must alter any existing the proxy relations involving the pending owner. First, the stealing task destroys all proxy relations between waiters on the lock and the pending owner. Second, the stealing task re-creates all proxy relations between waiters on the lock and itself.

It is possible that a lock be stolen from an existing waiter. This is a special case taken care of by treating the stealing task no differently than another waiter. The result is a configuration of data structures that represent the new owner being a waiter on itself. However, this inconsistency is resolved before the acquiring task re-enables interrupts and releases the spinlock protecting access to the RT-Mutex.

Waiter Interruption

Interruptable mutex operations allow a task to abort acquisition due to timeout, or the delivery of a signal. However, such an interruption can cause a locking chain to be "broken" at an arbitrary location. When an interrupted task is positioned at an edge of blocking graph (i.e. it has no waiters itself) only the interrupted task's proxy relation is removed. If the interruption occurs within a locking chain the proxy relation of each waiter on the interrupted task must be updated. Specifically, the interrupted task will become the new proxy of each of its waiters, and it is no longer blocked on a mutex.

Unlike the extensions to other RT-Mutex operations, this operation will constrain concurrency within the locking chain being broken. A waiter that is interrupted examines all of its waiters, removing any blocking relation between a waiter and a task up-stream the locking chain from the point at which the chain is being broken.

4 Evaluation

In this section the Group Scheduling framework is evaluated in terms of its generality, and the performance of its implementation. It may be impossible to deliver a formal proof of complete generality for a framework such as Group Scheduling, thus we opt for a showcase of schedulers implemented within the framework. The schedulers chosen are illustrative of the wide variety of semantics supported by the framework including classical schedulers, as well as some exotic breeds. Finally we offer a brief overview of the performance characteristics of Group Scheduling.

4.1 Balanced Progress SDF

We have developed a *balanced progress* SDF that schedules its members in such way that the *progress* of each member does not exceed that of other any other members, within a certain threshold. We use the term *progress* in a general sense, and it may take on different meanings in different applications. The specific application we describe here is the balanced progress of multiple processing pipelines, where progress is defined by the number of data units processed by a pipeline. For example this may be the balanced production of video frames.

GS Hierarchy

Figure 1 shows a typical Group Scheduling hierarchy used to implement an application with balanced progress semantics. At the root of the scheduling hierarchy is the *Sequential SDF* that schedules its members in the sequence they appear within the group. This is analogous to static-priority semantics. Its use here is to provide the computations under control of the *balanced progress* SDF preference over all other system computations, denoted generally as the *Linux* group.

Each pipeline in Figure 1 is shown as a group of computations controlled under a *sequential SDF* group. Each group of computations representing a pipeline is in turn a member of the *balanced progress* group. Each of the *balanced progress* members, denoted P1 and P2, represent a pipeline whose progress is to be balanced against other members of the *balanced progress* group. The scheduling parameters associated with each member in the *balanced progress* group is an integer value representing that pipeline's progress. When a pipeline completes a unit of computation it notifies the *balanced progress* scheduler to update the scheduling state. The implementation of the scheduling decision function that forces progress to never be out of sync by more than one unit of computation is shown in Program 4.1.

Program 4.1 Balanced Progress SDF

```
1  balanced_progress_choose_next(group) {
2     if (progress_is_equal(group))
3        return choose_member(group)
4     else
5        return least_progress(group)
6     endif
7  }
```

4.2 Guided Execution

The *guided execution* programming model was created for what we refer to as *deterministic concurrency testing* in which we provide a method for *guiding* a set of processes (or threads) into specific execution states in order to create a desired interleaving. A similar technique is used by the rt-tester to test execution scenarios *within* the RT-Mutex framework that are specified as a schedule of operations. The *guided execution* programming model is a generalization of the same concept, and can be applied to arbitrary codes using schedules that cross the user/kernel space boundary.

SDF Overview

The Guided Execution SDF schedules members according to a user-specified sequence of *execution contexts*, where an *execution context* is a tuple consisting a thread identifier and an opaque context object describing a state of execution for the thread. An example of such a schedule that guides three threads into a specific interleaving is:

```
1 (top):   (Thread-A, Context-A1)
2:         (Thread-B, Context-B1)
3:         (Thread-C, Context-C1)
4:         (Thread-B, Context-B2)
```

The SDF treats the sequence of execution contexts as a stack where the top of the stack is the *target* context. The SDF executes a schedule by choosing the thread associated with the target context. Once a thread has reached its target context the stack is popped and the next thread in the sequence is scheduled. A thread is chosen *only when* it is associated with the target execution context. In this way a set of threads can be guided into arbitrary interleaving.

In the above example Thread-A is first chosen to run until it reaches Context-A1. When the target context has been reached the stack is popped and Thread-B is chosen to run until it reaches Context-B1. The schedule continues to execute until all threads are in the context associated with their last appearance in the schedule. For example, Thread-B is first guided into Context-B1, and finally reaches Context-B2 at which point the schedule has been completed.

Guided Execution Programming Model

To demonstrate the use of the Guided Execution SDF we have built a programming model around the scheduler that expresses an execution context in terms of specific locations within source code. We refer to these locations as *way-points*. A way-point is a wrapper around the Group Scheduling API that is used to inform the Guided Execution SDF of a thread's current context. Our current implementation requires that code be modified by manually inserting way-points, but nothing will prevent way-points from being inserted using automated techniques such as automatic compiler insertion.

4.3 Other Semantics

Many other semantics have been created using the Group Scheduling framework, and both existing scheduling classes in Linux, real-time static priority and CFS are portable to the Group Scheduling framework. Notable schedulers implemented in the Group Scheduling framework include an *Explicit Plan* scheduler that is similar to the *Guided Execution* scheduler, but uses an explicit schedule of execution periods placed on a time-line. For example, it is trivial to implement periodic execution of threads implementing a work-loop based programming pattern.

Another scheduler with complex semantics implemented within the Group Scheduling framework is the PTides programming model developed at UC Berkeley as part of the Ptolemy group. This scheduler implements an actor-based model where actors communicate using timestamped events. A model implements a specification of a discreet event model. The scheduler is responsible for scheduling an actor to receive an even only when an event is safe to process. Portions of the safe to process analysis can be done statically, and at run-time sensors and actuators relate model time to physical time.

5 Conclusions and Future Work

We have presented Proxy Execution, our extension to the Group Scheduling that supports a general integration of scheduling semantics with concurrency control. While Group Scheduling itself facilitates the creation of arbitrary programming models, Proxy Execution full integrates the scheduling semantics of such models with the concurrency primitives in a system, creating a truly complete solution.

The use of Group Scheduling can reduce costs by easing the implementation of complex application semantics, and increases the accuracy and understandability by using direct rather than indirect implementations. Finally, Group Scheduling can be utilized as a framework for comparison of different scheduling algorithms and policies. Implementing new ideas in a single framework allow a fair comparison as tests can be performed using identical system configurations and hardware profiles.

Further information and links to our software can be found at the KUSP website: *http://www.ittc.ku.edu/kusp/*

References

[1] Liu, C. L. and Layland, James W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment *Journal of the ACM*, 1973.

[2] Lehoczky, J., Sha, L., and Ding, Y. The rate monotonic scheduling algorithm: exact characterization andaverage case behavior *Real-Time Systems Symposium*, 1989.

[3] Tejasvi Aswathanarayana. Integrating concurrency control & proxy execution support and provide a framework for deterministic concurrency testing under the kurt-linux group scheduling model. Master's thesis, University of Kansas, September 2001.

[4] Steven Rostedt and Darren V. Hart. Internals of the RT Patch. *Proceedings of the Linux Symposium*, June 2007.

[5] Tejasvi Aswathanarayana, Douglas Niehaus, Venkita Subramonian, and Christopher Gill. Design and performance of configurable endsystem scheduling mechanisms. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:32–43, 2005.

[6] Michael Frisbie, Douglas Niehaus, Venkita Subramonian, and Christopher Gill. Group scheduling in systems software. *Parallel and Distributed Processing Symposium, International*, 3:120a, 2004.

[7] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.