A Practical Framework for Configuration of Scheduling and Concurrency Control in Linux

By

Tyrian Phagan

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Chairperson Douglas Niehaus

Perry Alexander

Prasad Kulkarni

Date Defended: 6/23/11

The Thesis Committee for Tyrian Phagan
certifies that this is the approved version of the following thesis:

A Practical Framework for Configuration of Scheduling and Concurrency Control in Linux

Chairperson Douglas Niehaus

Date Approved: 6/23/11

Abstract

There is a growing need for developers to be able to specify programming models for an application, in order to: increase efficiency, system reliability, system security, and to allow applications with different semantics to coexist on the same system. Only specifying the scheduling semantics for an application is not sufficient because concurrency control also significantly affects the behavior of the application. This work demonstrates the integration of the Hierarchical Group Scheduling and Proxy Management frameworks to provide the ability to developers to configure scheduling and concurrency control semantics for a wide range of applications. This work targets the Linux platform to be useful to a large audience of developers. Additionally, an environment for verifying the correctness of this integration and other concurrent applications using deterministic testing is discussed.

Acknowledgments

I would like to thank Douglas Niehaus, my advisor and committee chair, for providing guidance during the work presented here. I would also like to thank Perry Alexander and Prasad Kulkarni for serving as members of my thesis committee. I would like to acknowledge Noah Watkins and Jared Straub for providing the implementation upon which this work was based. Finally, I would like to recognize the National Science Foundation for providing the funding for this project through grants CCF-0615035 and CNS-0716740.

Contents

List of Definitions

List of Figures

List of Programs

Chapter 1

Introduction

There is a growing need for applications, coexisting on the same system, to have different scheduling semantics. Reasons for this include: (1) economic pressures to combine applications on a single computer rather than using separate computers, (2) a desire for more precise control of application behavior on the part of many developers, and (3) a desire for more precise control of system and application behavior to increase efficiency, system reliability, and system security.

Many projects address scheduling in one form or another but the problem is hard enough that in most cases the approach is not sufficiently comprehensive to be effective outside a restricted application domain. Most people do not have sufficient expertise, or it is not reasonable, for them to change the operating system and so most approaches to specialized scheduling use concurrency control to achieve desired effects. This can be effective in some cases but in many cases it is not as accurate nor as efficient as desired. For instance, the vast majority of systems only provide priority based scheduling semantics and developers must use priority manipulation and concurrency control to adapt these semantics to produce the desired application behavior.

Scheduling parameter manipulation alone is insufficient because concurrency control has such a strong effect on application behavior. Indeed, one aspect of this fact is the frequency

with which concurrency control is used in applications for scheduling effects rather than to protect critical sections. Thus, even though it is extremely difficult, any effective approach to flexibly configurable scheduling semantics must include full integration of scheduling and concurrency control.

Mainline Linux is predominantly concerned with support for applications which perform adequately under conventional priority based scheduling, under concurrency control using FIFO semantics, and which emphasizes average case performance of the system as a whole. However, there are an increasing number of growing user communities for which conventional scheduling and concurrency control semantics are not adequate. Among the most obvious examples of this are the several categories of real-time systems in which the standard semantics for both scheduling and concurrency control are unacceptable. In the scheduling domain, standard Linux provides simple real-time scheduling in the form of round-robin and FIFO, but provides only FIFO concurrency control semantics. Unfortunately, rate-monotonic analysis, among the most popular of real-time scheduling approaches, requires Priority Inheritance support in concurrency control. For this reason, the PREEMPT-RT patch provides Priority Inheritance for kernel mutexes and user-level PI-futexes [17].

Other approaches to real-time do not necessarily want to use priority scheduling or Priority Inheritance and thus require additional modifications beyond those of the PREEMPT-RT patch. In addition, any modification to scheduling semantics must also modify concurrency control semantics if FIFO concurrency control is not appropriate. There are several ongoing attempts to add different scheduling semantics to Linux [9] [4]. These are all heavyweight modifications because they require extensive modification to core operating system software. There is no support for dynamically adding new scheduling semantics to the system under standard Linux or PREEMPT-RT. A reasonable definition for dynamic is that the scheduling semantics can be changed under kernel and program control at runtime according to user specified configuration.

There are other important application areas in addition to real-time that either currently

desire or would benefit from flexibly configurable application semantics. Large scale scientific computing, various varieties of embedded systems, frameworks for testing concurrent software, and different types of multimedia systems all have semantics that could benefit from the ability to flexibly configure the control semantics of their computations. While many efforts have existed over the years to improve program behavior semantics in various ways, most have suffered from the handicap of being partial solutions [1] [16] [4]. Some change scheduling without affecting concurrency control and others change concurrency control without affecting scheduling. Others have tried to integrate concurrency control and scheduling but virtually none provided a feasible way to flexibly configure application semantics across as wide a range as the method presented in this thesis.

We believe that the combination of Hierarchical Group Scheduling (HGS) and Proxy Management (PMGT) represents the first sufficient approach to the problem of flexibly configurable application control semantics. PMGT provides an accounting method independent of scheduling semantics which is used by mutex concurrency control and which can support the system scheduling layer with a configurable interface. This thesis demonstrates that the PMGT interface can be used to support both the scheduling layer in PREEMPT-RT and to support more directly specified and customized application semantics in HGS.

HGS supports a hierarchical representation of system scheduling semantics. This allows applications with different semantics to co-exist on the same system because the scheduling hierarchy for the system describes how conflicts among applications should be resolved. In addition, no permanent modifications to the system are required for an application to create new semantics. Instead, the application can define a customized programming model by creating a customized scheduler, creating customized application libraries, or by creating customized concurrency control policies. The configuration of the system is then modified to include the application of these customized policies to specific sets of applications. HGS also allows an application to manipulate its own semantics at runtime. The combination of HGS and PMGT thus provides a much broader range of configurable behavior beyond what

is possible with the current selection of specialized heavyweight solutions.

This thesis also presents a customized programming model called Guided Execution which was developed as an environment within which the correctness of PMGT and HGS implementations could be tested. Guided Execution implements a customized scheduler and controls the threads implementing the tests under the HGS framework just as any other application would specify and use a customized semantics. Guided Execution provides an environment in which deterministically testing the correctness of PMGT and HGS using a set of over 400 scenarios involving concurrent execution of multiple threads is possible.

1.1 Contributions of this Thesis

The work presented in this thesis was based on a large body of work, some of which was already partially done when the work described here began. The primary contribution of this thesis was to fully realize the integration of mutex concurrency control with scheduling. This thesis also demonstrates that the integration of concurrency control and scheduling is complete and correct in several ways.

The major contributions of this thesis include:

1. Significantly restructured the original PMGT implementation to improve performance and reliability.

2. Creation of the ability for PMGT to support scheduling layers other than HGS and separation of the scheduler API from the mutex API.

3. Creation of a Priority Inheritance implementation that utilizes PMGT which demonstrates that the new framework can reproduce the previous semantics correctly.

4. Creation of the Exclusive Control Scheduling class to properly integrate HGS into the existing Linux scheduling framework, eliminating instability and incorrectness plaguing the previous implementation.

5. Extension of basic HGS mechanisms from the uniprocessor to the multiprocessor domain.

6. Creation of a method for managing proxy relations that cross the boundaries of all scheduling classes in Linux: HGS, real-time, and the completely fair scheduler.

7. Realization of the Guided Execution scheduler for deterministic testing of scenarios involving concurrency. Specifically, this was used to implement the 31 tests required to demonstrate proxy accounting correctness and more than 400 tests to demonstrate HGS correctness.

The rest of this thesis first discusses various approaches to scheduling and the testing of concurrent software that are related to the contributions of this thesis in Chapter 2. Chapter 3 discusses background necessary to understand the framework within which the contributions of the thesis were implemented, the PMGT and HGS implementations, and then discusses the Guided Execution programming model and its use in demonstrating the correctness of the PMGT and HGS implementations. Chapter 4 first discusses the basic design and methods used for the evaluation experiments and then presents experiments evaluating the performance of PMGT and HGS in various ways. Finally, Chapter 5 presents conclusions and discusses future work.

Chapter 2

Related Work

The work presented in this thesis primarily deals with two issues: the integration of scheduling and concurrency control and the testing of that integration. The first half of this section presents an overview of several different scheduling implementations and their treatment of the integration of scheduling and concurrency control. The second half of this section examines several different methods that might be applied to test concurrent programs that utilize the integration of scheduling and concurrency control.

2.1 A General View of Scheduling Layers

A scheduling algorithm provides the method by which application behavior can be specified and by which that behavior can be produced as it executes. A scheduler is an implementation of an algorithm and executes in the system as the controller of application execution.

The behavior produced by a particular execution of a scheduler depends on the set of entities controlled by the scheduler and the state data associated with those entities. Generally, a group is the set of entities, either groups or tasks, controlled by a scheduler and the state data associated with those entities. A scheduler is associated with each group and makes use of the group's state data in making its decisions.

A scheduling layer provides a set of standard capabilities to make writing a scheduler easier and more reliable. A scheduling layer which permits more than one scheduler to be used concurrently must also provide ways for the developer to specify which tasks are controlled by which schedulers and to resolve conflicts among the various schedulers. The scheduling layer must ultimately provide a way to combine the semantics of the various schedulers to produce a unified scheduling semantics for the system.

With respect to the issues addressed in this thesis, there are four important characteristics of scheduling layers: (1) support of common requirements of most or all schedulers, (2) the ability to organize a set of schedulers, (3) dynamic configuration, and (4) integration with concurrency control.

Support of Common Scheduler Requirements

Each scheduling layer assumes a set of requirements that are common to all schedulers. Most scheduling layers implement their treatment of the common requirements as a framework using a set of function pointers to represent components of a generic scheduler. This both requires and allows all schedulers implemented under the framework to have a similar structural decomposition as a set of functions. Each function pointer represents a component of the common model invoked in situations where that component function of the scheduler is required to act, in the view of the common scheduling layer.

While each scheduling layer differs from others in detail, the function pointers used to implement a model can be broken into a number of common categories: those invoked at scheduling time, those that change the state of a task, those involved in load balancing, those invoked in the context of the system timer tick, and those supporting integration of scheduling and concurrency control. Note that a given scheduling layer may not implement function pointers in every category.

Organizing Multiple Schedulers

Multiple scheduling algorithms may be required to specify the behavior of a system. Each scheduling algorithm can be simplified if the algorithm does not have to specify its interaction with every other algorithm. Therefore, the scheduling layer is often used to implement a unifying supervisor that specifies how the scheduling algorithms interact. Such a unifying supervisor is a reasonable addition to a scheduling layer because it is obvious that portions of the system controlled under different scheduling semantics may come into conflict. If a scheduling layer does not explicitly represent how such conflicts are managed then the semantics are left implicit because every scheduling layer must ultimately choose a single thread to run on a given CPU at any given time.

Dynamic Configuration

If the set of schedulers can be modified and organized dynamically at runtime then the system can better support application-specific scheduling semantics by permitting applications to implement their own scheduler. Otherwise, the system must know beforehand every set of control semantics that will be required. On a general purpose system, where the applications on the system may change constantly, this is very difficult to know. On an embedded system, configuration of the set of schedulers at system generation time could be feasible.

Integration with Concurrency Control

Currently, on most general purpose systems, applications must use a combination of priority assignment and concurrency control primitives to implement desired behavior semantics. This is often less precise, more difficult to implement, and more difficult to understand than the explicit specification of the behavior semantics as a scheduler.

Integration of scheduling with concurrency control is important because the concurrency control affects the runnability of a thread. If the semantics of the concurrency control are at odds with the semantics of the scheduler then the application behavior produced may

diverge significantly from that intended. There are two obvious types of integration: (1) waiter selection and (2) influence of the waiters on owner scheduling. The first refers to how a task waiting for a mutex is selected when the mutex is released by its owner. The second refers to whether the scheduling semantics of the waiters should influence how the owner is treated by the scheduling layer.

As an example of the need for waiter selection integration, imagine a system using priority scheduling but FIFO semantics for mutexes. The task chosen to receive ownership of a mutex might not be the task with the best priority which would be contradictory to what the scheduler would choose. In contrast, if priority is used to choose a waiter then the concurrency control and scheduling semantics are integrated.

The original form of Rate Monotonic Analysis (RMA) and its subsequent extension illustrate the need for the integration of concurrency control and scheduling.

Originally, RMA did not consider the effects of resource use, simply assigning priority according to the frequency with which periodic tasks were executed [14]. However, under this approach, a task with a better priority could wait an arbitrary period of time while tasks with worse priorities executed because of a lack of mutex ownership and scheduling integration. This is commonly known as the Priority Inversion problem. While it was originally identified in the context of RMA, it is applicable to any priority based scheduling. Priority Inheritance is the most widely accepted solution which influences the scheduling semantics of the mutex owner by making its dynamic priority the maximum of its own and its waiters [18].

It is interesting to consider the possibility that under a framework where any concurrency control semantics can be specified then using concurrency control primitives for scheduling effects could in principle no longer be required.

Summary

Any scheduling layer will be based on some view of what the fundamental aspects of scheduling are and this view will strongly influence the basic architecture and semantics of

the framework. To illustrate this we consider five scheduling frameworks: (1) the "scheduling stack" in current standard Linux, (2) the Hierarchical Loadable Scheduler, (3) the original form of HGS which was called Group Scheduling at the time, (4) the PREEMPT-RT patch for Linux, and (5) Litmus-RT.

2.1.1 Linux Scheduling Stack

The standard Linux Scheduling Stack(LSS) implements several different scheduling classes. A scheduling class is the implementation of one or more scheduling algorithms. In Linux, each scheduling algorithm is referred to as a policy. Conceptually, a policy is the same as a scheduler and the term scheduler will be used here for consistency.

The LSS partially fulfills the first criteria of an effective scheduling layer by providing a set of twenty callbacks that can be used to implement a scheduling algorithm. Most of these callbacks would be useful to schedulers with a wide range of semantics. However, Linux relies on the **sched_setparam** and **nice** system calls for changing the scheduling parameters of a task and these have no support for non-priority based scheduling parameters. The LSS also does not provide callbacks for integrating scheduling with concurrency control.

The LSS organizes the scheduling classes into a *stack* that determines the order in which the classes are evaluated. Each scheduling class determines the ordering of its schedulers. As a result, the order of the classes in the stack and the order of the schedulers within the classes determines a total order on all schedulers in the system and thus a unified semantics for the system.

In mainline Linux, there are three scheduling classes: Real-Time, the Completely Fair Scheduler (CFS), and Idle. This is the order of the evaluation of these classes in the Linux scheduling stack. The Real-Time class implements First-In-First-Out (SCHED_FIFO) and Round Robin (SCHED_RR) static priority schedulers. The CFS class is the class used by most threads under Linux. It implements the Other scheduler (SCHED_OTHER) that is priority based but adjusts dynamic priority in response to recent CPU consumption. CFS

also implements the Batch scheduler (SCHED_BATCH) for low priority, long running tasks. The Idle class implements only the Idle (SCHED_IDLE) scheduler that provides the a CPU with busy work when there is nothing else to do.

The set of scheduling classes in the stack and their order is static but Linux has some support for dynamic configuration of groups. A hierarchy of groups can be created within each of the scheduling classes called *Cgroups* [6]. A scheduling class uses the same scheduling algorithm for all of the groups in its hierarchy and a task can only be a member of a single group. Primarily, the hierarchies are used to specify how CPU resources are divided for each class. While this model provides considerable range for configuring hierarchies of resource related scheduling constraints, it fails to provide the range of configurability for scheduling semantics provided by other frameworks such as HLS or HGS which are discussed later in this section.

Recently, Linux has added the ability to create groups of tasks under Cgroups based on their login session (TTY) [7]. This has improved the responsiveness of the Linux desktop because all desktop tasks can be made more important than background processes. This new feature also makes the LSS more dynamically configurable without going so far to support dynamically changing sets of schedulers.

The LSS exclusively cares about priority scheduling and so some of the features that might be useful to a scheduling layer that allows a wider range of scheduling semantics are less important in achieving its desired system semantics. The lack of support for dynamically changing the set of schedulers and organizing the order in which schedulers are applied does not significantly hinder the goals of the LSS.

Finally, Linux does not provide any integration with concurrency control. All Linux concurrency control uses FIFO semantics. Clearly this limits the range of available semantics and some extension in this area is desirable. The PREEMPT-RT patch and HGS discussed later in this section address extension in this area in different ways.

2.1.2 Hierarchical Loadable Scheduler

The Hierarchical Loadable Scheduler(HLS) was developed as a scheduling layer for Windows 2000 [16]. In many ways, HLS is similar to HGS. The primary difference and disadvantage of HLS compared to HGS is the lack of integration between scheduling and concurrency control.

Scheduling algorithms are implemented in HLS using a set of callbacks. However, HLS uses callbacks only invoked at scheduling time and to set the scheduling parameters of a task. Since this was a research project, Reghr chose to concentrate on composition of scheduling semantics and did not include more programmatic aspects of other frameworks such as timer tick or load balancing callbacks.

A central theme of HLS is that the system behavior is specified by a hierarchy that composes different scheduling semantics. Under HLS, groups are called "scheduler instances". For consistency, the term group will continue to be used here.

An HLS hierarchy consists of groups, tasks, and virtual processors. Under HLS, a task can only be a member of a single group but it is still possible to compose scheduling semantics. A "Join" HLS scheduler exists that enables a task to be influenced by multiple groups. A group assigned to the Join scheduler contains a single entity, either a task or a group, and always schedules that entity whenever it is asked to make a decision. Scheduling semantics can be composed by placing a task under the control of an instance of the Join scheduler and placing that instance under the control of multiple groups.

In an HLS hierarchy, all connections between groups are made through virtual processors. When a scheduling decision needs to be made for a particular physical processor then the top group in the hierarchy assigns the physical processor to a virtual processor it controls. The assignment of a physical processor to a virtual processor gives processor time to the groups connected to that virtual processor. The virtual processor scheme makes scheduling involving processor resources explicitly configured in the HLS hierarchy.

HLS allows the set of schedulers to be modified dynamically and the hierarchy can be

re-organized. Therefore, HLS is well suited for general purpose systems that have many applications with different scheduling semantics because the system does not have to implement the full set of schedulers for these scheduling semantics at build time.

HLS does not provide any integration with concurrency control. Instead, it is recommended that threads should avoid situations that would require concurrency control support, such as priority inversion. Avoidance, when possible, requires more effort by application developers than the direct concurrency control integration provided by HGS. Additionally, avoidance may not always be possible.

Time-sharing, fixed priority, and CPU reservation scheduling algorithms have been implemented using HLS.

2.1.3 Group Scheduling Without Concurrency Control Integration

The original implementation of HGS was simply called Group Scheduling(GS) and implemented only the scheduling aspects of HGS without concurrency control integration [1]. One of the primary contributions of this thesis was to complete the addition of support for the integration of concurrency control and scheduling.

GS provided common support for schedulers by establishing a set of callbacks implemented by all schedulers which included getting and setting arbitrary scheduling parameters. However, it did not provide a load balancing callback, a timer tick callback, or callbacks used to integrate with concurrency control.

GS uses a hierarchy consisting of tasks and groups to specify system behavior. Both tasks and groups can be a member of one or more groups. A scheduler is associated with each group and chooses among group members when invoked. A given scheduler can be associated with more than one group but since each group's member data is used separately, this is best viewed as multiple instances of the same algorithm being associated with different groups.

At scheduling time, the scheduler associated with the top group of the GS hierarchy is invoked. It may choose to query a group which is one of its members. This query takes the

form of a nested call to the subordinate group's scheduler. At the level of a given group several queries to subordinate groups may be required. As a result, the pattern of execution within the scheduling hierarchy is one of a recursive search through a decision tree until a task is picked.

While both GS and HLS use a similar hierarchical description, there are a few practicalities that differentiate the two. One of the significant differences is related to the idea of virtual processors and is related to the difference in target platforms for GS (Linux) and HLS (Windows). Under HLS, tasks associated with a virtual processor must be executable on any physical processor assigned to the virtual processor. However, under Linux, tasks can only be executed on a specific assigned processor and changing the processor assignment involves significant overhead. Thus, the virtual processor approach of HLS would be too expensive under Linux and GS used a different approach. An additional difference is the use of the Join scheduler by HLS as a mechanism used to permit tasks to be controlled by more than one group. Under GS, tasks are simply permitted to be members of multiple groups and therefore these semantics are represented directly.

GS provided dynamic configuration. Schedulers could be added and removed from the system at runtime and the association between schedulers and groups could also be dynamically adjusted. Thus, GS is easily applied to general purpose systems with a dynamic set of applications that desire a wide range of behavior.

The desirability of using application-specific customized scheduling under this framework was demonstrated for a simulated video processing application that used multi-threaded pipelines for each video stream and wanted to maintain balanced progress of the processing for multiple streams [1]. However, this application did not involve uses of concurrency control where integration with scheduling was relevant to the performance metrics. Therefore, the results were excellent even though the framework had a significant limitation. Recognition of the need for the integration of concurrency control and scheduling motivated the development of HGS.

2.1.4 Real-Time Preemption Patch

The Real-Time Preemption Patch (PREEMPT-RT) improves on the LSS by providing integration of scheduling and concurrency control [17]. PREEMPT-RT also improves the overall control in the system by providing a thread context for hard-irqs and soft-irqs, thus including essentially all system computation activities in a unified scheduling framework.

In most respects, PREEMPT-RT has the same characteristics as the LSS: (1) it is exclusively interested in priority scheduling, (2) it provides a set of function pointers that are used by schedulers to implement scheduling algorithms but PREEMPT-RT assumes only priority based schedulers will be implemented, (3) the set of schedulers for the system are organized in a static stack, and (4) there is limited dynamic configuration.

PREEMPT-RT addresses the fourth criterion for an effective scheduling layer, the integration of scheduling and concurrency control, by providing support for Priority Inheritance. This approach to integration is limited to schedulers using priority semantics. Nonetheless, it is an important milestone because it makes it possible to support a number of popular real-time and other priority based scheduling semantics in standard Linux. PREEMPT-RT addresses the waiter selection issue by replacing the FIFO waiter selection policy with a priority based one. Its implementation of Priority Inheritance permits the set of waiters for a mutex to influence the scheduling semantics of the mutex owner.

PREEMPT-RT is an appealing platform for non-priority based scheduling due its improved control over all system computation activities. However, in most cases, some extension is necessary to overcome the limitations of the LSS upon which PREEMPT-RT is based and also to generalize the concurrency control semantics provided by PREEMPT-RT to work with a wider range of scheduling semantics. Litmus-RT, HGS, and efforts to implement deadline scheduling [9] are examples of work based on the PREEMPT-RT patch but attempting to extend it in various ways.

2.1.5 Litmus-RT

Litmus-RT is a scheduling framework specifically targeting real-time scheduling algorithms [4]. Litmus-RT is a modification to PREEMPT-RT. It provides a new scheduling class that is placed at the top of the LSS. Litmus-RT is placed at the top of the LSS to make it the scheduling class consulted first when making a scheduling decision. This technique is also used by HGS.

The Litmus-RT scheduling class provides a plugin framework within which a variety of real-time schedulers can be implemented. It provides its own API which is somewhat simpler than that of the LSS API. While Litmus-RT is a scheduling class from the LSS's point of view, it is probably more accurate to consider it a scheduling layer.

Litmus-RT, like most scheduling layers, assumes a common model for schedulers used to implement each scheduler. The Litmus-RT plugin framework provides function pointers invoked (1) at scheduling time, (2) when a timer tick occurs, (3) when scheduling parameters are changed, and (4) for a limited form of concurrency control integration. The scheduling parameter function pointers allow non-priority parameters to be specified.

The second criterion for an effective scheduling layer, the ability to organize a set of schedulers, is not addressed by Litmus-RT, because it only permits a single scheduler to be active at any time. It does, however, provide some dynamic configuration by permitting several schedulers to be included at compile time and allowing the active scheduler to be switched at run time. Also, Litmus-RT does not allow addition of new schedulers at runtime.

Several Litmus-RT schedulers implement scheduling algorithms that require different strategies for resource use. Litmus-RT specifies a basic strategy for resource use called the Flexible Locking Multiprocessor Protocol (FMLP) which is modified for each scheduling algorithm [2]. The implementation of FMLP shares the basic strategy among the schedulers and integrates scheduling with concurrency control by calling into the active scheduler to allow modifications to the basic strategy to be made.

FMLP classifies resources into resource groups. A resource group is either "short" or

"long" and each resource group is protected by a resource group lock. A short resource group is controlled by a FIFO busy-wait lock that requires that the owner be non-preemptable while the lock is held. The locks for short resource groups are not integrated with scheduling for the obvious reason that they are non-blocking. A long resource group is controlled by a Litmus-RT semaphore implementation which has a FIFO queue for waiters. The owner of the semaphore inherits the best priority of the waiters, creating a strange hybrid of FIFO and PI semantics. Another notable aspect of the PI implementation is that it appears to only consider those tasks directly waiting on the semaphore and not those indirectly waiting through other semaphores when determining the inherited priority. Additionally, this implementation calls into the active scheduler when (1) a task blocks on the semaphore, (2) the owner inherits a priority, and (3) the semaphore is released to allow the scheduling plugin to update scheduling data. These callbacks seem to be used to effect preemptability and to adjust priority of threads but precisely why this would be useful in a particular situation was not clear [2].

Though FMLP is still restricted to specific concurrency control semantics, it demonstrates an approach to the integration of concurrency control and scheduling that communicates more information to the scheduling layer than the simple priority values used in the PREEMPT-RT implementation of PI.

HGS integrates with the system's concurrency control using a similar strategy to Litmus-RT by placing callbacks in the modified mutex implementation provided by PMGT. However, unlike Litmus-RT, HGS makes few assumptions about the scheduling semantics of the tasks using a mutex by making waiter selection semantics configurable on a per mutex basis and generalizing the influence of waiters on the scheduling of the mutex owner. Additionally, the concurrency control semantics configured by HGS apply to all of the system resources in order to provide greater control over system behavior. The intent is to permit an application to create specialized concurrency control semantics for an application just as it can create a specialized scheduler for controlling application execution.

2.2 Testing of Concurrent Software

The increasing popularity of multi-core processors has made multi-threaded concurrent software ever more important. However, concurrent software is difficult to develop. Part of the difficulty lies in testing the correctness of the software because, under most programming models, the execution interleaving of threads in concurrent software is non-deterministic.

In testing concurrent software, the primary concern is the order of synchronization operations performed by the threads executing the software. Synchronization operations are those operations that interact with another thread, such as: reading from a communication channel, writing to a communication channel, a lock operation on a semaphore, or an unlock operation on a semaphore [13]. The sequence of synchronization operations is known as a SYN-Sequence [13].

To adequately test a concurrent piece of software, every possible SYN-Sequence must be considered. The set of SYN-Sequences can sometimes be reduced by removing similar sequences in order to simplify testing. Therefore, a testing framework must be able to insure that a desired set of SYN-Sequence has been tested by either observing that each SYN-Sequence has executed or by forcing every SYN-Sequence to execute.

Some tools that are generally useful for testing are not useful for concurrent testing. Code Coverage Analysis is one method that is helpful for testing software in general but gives little insight into the concurrent aspects of the software. This method can provide information about which sections of the software have been executed and which branches were taken [3]. However, this information is not of much use for testing concurrency because it does not indicate which threads executed which synchronization operations and does not provide the order in which the synchronization operations occurred.

Four ways of testing concurrent software are (1) non-deterministic testing, (2) deterministic testing, (3) reachability testing, and (4) model checking.

2.2.1 Non-deterministic Testing

Non-deterministic testing executes the same program many times with a fixed input in an attempt to cause all of the SYN-Sequences to occur [8] [19]. In this type of testing, there is no control over which SYN-Sequence is executed. Instead, the testing narrates which SYN-Sequence was executed. In addition, non-deterministic testing can generate SYN-Sequences without the SYN-Sequence first being specified. Thus, it can serve as a good starting point to determine which SYN-Sequences are possible.

In practice, not all test suites have a way to know which SYN-Sequence occurred and it is assumed that if a large number of tests ran successfully then all SYN-Sequences occurred. However, this form of testing is dangerous because there is no way to guarantee that all of the SYN-Sequences have in fact occurred.

Often, non-deterministic testing relies on adding random noise to a program in order to cause a sequence to occur but this is not precise and it does not guarantee that a given sequence will occur. It thus provides no guarantee how long it might take for a give non-deterministic test suite to cover every required SYN-Sequence.

2.2.2 Deterministic Testing

Deterministic Testing forces a program to execute a specific SYN-Sequence [5]. Using Deterministic testing, any given SYN-Sequence is easy to execute. In this respect, it is has a significant advantage over non-deterministic testing. However, every SYN-Sequence must be explicitly specified. Often, in order to insure that every SYN-Sequence has been specified, a static model must be constructed. The primary disadvantage of deterministic testing is that the model that is constructed may be complicated.

Deterministic testing is either implementation based or language based [5]. Implementation based testing modifies the compiler, middleware, operating system, or hardware [5]. Middleware modification is the most popular approach.

Deterministic testing relies on controlling the interleaving of the threads that are exe-

cuting the software. Compiler modifications do not directly influence or effect how the set of threads executing are interleaved. Thus, while compiler modifications may be useful in a particular approach by permitting enhanced instrumentation or modifying the semantics of concurrency control to aid testing on their own they are unlikely to support a complete solution.

Middleware modifications may be the most popular because it is the most accessible for modification by those creating the test framework rather than being the best place to accomplish specific purposes, necessarily. In testing that uses middleware, the middleware layer re-implements all the synchronization operations involved in the tests [13]. The new synchronization operations constrain task execution to adhere to a specified SYN-Sequence using concurrency control. A major advantage of modifying the middleware is that no modifications to the operating system are required. Modifications to the operating system often require greater expertise, more effort, and access than the test framework developers possess.

Hardware support has also been used to provide support for deterministic testing. However, often, specialized hardware is expensive in terms of both money and development time. An approach that works on general purpose systems is preferable. One such hardware approach is called SMILE [12]. SMILE uses hardware monitoring to intercept memory accesses, queue the memory accesses in a transaction pool, and then execute the memory accesses in a specific order. Thus, SMILE is good for executing SYN-Sequences if the synchronization operations are all accesses to shared memory. This approach is limited as a general testing approach because no other types of synchronization operations can be tested.

Other deterministic testing methods use a language based approach. Language based approaches translate a non-deterministic program into a deterministic program [5]. Language based approaches provide more portability because the middleware does not have to be ported to a separate platform but some programs are too complex to be translated.

Operating system modifications are rare due to their increased complexity and a lack of

access available to most developers. Guided Execution (GE) provides a means to perform deterministic testing using a combination of operating system modifications and program translation. GE is based on directly scheduling task execution order instead of indirectly influencing it through concurrency control.

To test a program using GE, a program is translated to provide information to the system scheduler related to which task should execute at any time by adding control points to the program where context switching among threads may be required to implement a specific scenario. These control points are called waypoints and are implemented as calls from the application into the GE scheduler informing it that a specific task has reached a specific point in the application code.

GE provides direct control of which task is executing at any given moment by modifying how the system scheduler controls execution of test scenario threads. Specifically, the GE scheduler follows a specified test scenario schedule expressed in the form of a sequence of waypoints reached by specific threads.

The specific use of Guided Execution discussed in this thesis presents a special set of problems for traditional approaches which use concurrency control primitives in the testing framework. Specifically, we describe how we test both the user level PI-futexes and the system level mutex primitives. The two are closely related because whenever contention arises at the user level a system call is made which creates a system level mutex to represent the user level PI-futex. The problem with middleware based deterministic testing for this specific problem is that it would require using concurrency control primitives to test themselves. Guided Execution provides a control method outside the concurrency control primitives and thus is a significantly better approach for testing the concurrency control primitives themselves.

However, some system modifications are required for GE and GE is restricted to Linux. Middleware requires no system modifications which is clearly desirable when considering only the implementation of a testing framework. One limitation of the middleware approach

is that it must be possible to achieve the desired semantics through only modifying the concurrency control primitives involved. This assumes that the existing API provided by the system is sufficient to the task. Additionally, a purely translation based approach has the advantage of being essentially platform independent with the exception of relying on the concurrency control available on the platform to be sufficient for producing the desired SYN-Sequences.

GE is thus an example of deterministic testing which provides few if any new capabilities. However, it is interesting for 3 other reasons. First, it illustrates that a programming model with highly specialized semantics can be produced under the HGS framework in a straightforward way. Second, the GE model can be used to test the correctness of concurrency control primitives which might not be possible in a framework that depends on concurrency control primitives to implement the testing. Third, it illustrates that the presence of the HGS framework greatly decreases the effort of implementing operating system based methods of implementing deterministic concurrency testing. We believe that direct implementation at the system level is clearer and thus preferable to indirect methods using concurrency control for scheduling effects but that is arguably more a point of preference than capability.

2.2.3 Reachability Testing

Reachability testing is a combination of deterministic and non-deterministic testing. Reachability testing uses non-deterministic testing to find new SYN-Sequences and deterministic testing to execute specific SYN-Sequences that have been found [13]. Reachability testing aims to alleviate the need for the user to generate all SYN-Sequences as is required in deterministic testing.

In reachability testing, a program is first executed with a given input to generate a random SYN-Sequence. New SYN-Sequences are generated from this one. All prefixes of the original SYN-Sequence are then generated by removing operations from the end of the sequence. For every prefix generated, prefix-based replay is performed to generate a new

SYN-Sequence.

In prefix-based replay, a program is deterministically guided to the end of the prefix and allowed to execute non-deterministically thereafter. The SYN-Sequence generated during non-deterministic execution is recorded and appended to the prefix to create a new SYN-Sequence. The new sequence created can be used to generate other sequences using the same prefix generation and non-deterministic execution method. A dictionary is kept to prevent using a prefix more than once [13].

GE has not yet been used to perform reachability testing. However, GE is capable of performing prefix-based replay and it provides a narrative that could be used to indicate which synchronization operations occurred during the non-deterministic portion of the replay. Therefore, the basic tools necessary to implement Reachability Testing are present because new SYN-Sequences could be generated from existing sequences by performing prefix-based replay.

2.2.4 Model Checkers

Model checkers require a model of the software to be built and then the model is tested instead of the software itself.

One such model checker is called SPIN. To use SPIN, software semantics must be described using SPIN's modeling language PROMELA [11]. PROMELA can be used to fully specify an algorithm. It goes beyond considering just the synchronization operations. SPIN builds a state based model from the PROMELA specification and tests combinations of model states.

In addition to examining model states, SPIN can perform some basic safety checks of the model. The safety checks are similar to Code Coverage Analysis. SPIN also allows specific execution and algorithmic conditions to be specified that should never occur in the model [11]. If one of the conditions does occur then SPIN provides a report of the set of operations that led to that condition.

Model checkers such as SPIN provide additional flexibility. However, care must be taken to specify a correct description of the software semantics as a model. This can be difficult for large or complex software. Additionally, it may be difficult to know all of the conditions to specify that should not occur in the model. Finally, SPIN can create very large models that are expensive to evaluate [11].

In comparison, GE does not provide the automatic safety checks of a model checker. It also cannot check for the absence of specific conditions. However, by testing the implementation instead of the model, GE insures that nothing has been lost in the creation of the model and, thus, using model checkers in combination with GE may be beneficial.

Chapter 3

Implementation

This first half of this chapter discusses the background context that is necessary to understand the work presented in this thesis. Datastreams, an instrumentation framework, is discussed first in Section 3.1. It is used to verify and evaluate the work presented in this thesis. Next, the Computation Component Set Manager (CCSM) is discussed in Section 3.2. CCSM provides facilities for identifying and grouping tasks. Finally, the fundamentals of Proxy Management and Hierarchical Group Scheduling are discussed in Section 3.3 and Section 3.4, respectively.

The second half of this chapter explains the extensions made to Proxy Management and Hierarchical Group Scheduling as the main contributions of this thesis in Section 3.5 and Section 3.7, respectively. Finally, the creation of a programming model, called Guided Execution, for testing concurrent software and its use to create a test suite for PMGT and HGS are discussed in Section 3.8.

3.1 Datastreams

Datastreams provides flexible, unobtrusive instrumentation that can record arbitrary data. The kernel (DSKI) and user-side (DSUI) implementations have uniform interfaces to make the instrumentation clear and easy to use. Events are recorded in a binary format that

is later interpreted by a Python based post-processing. A post-processing phase is superior to formatting the data before recording the event because it decreases the instrumentation effect by requiring less processing during the experiment.

DSUI automatically spawns additional threads that write events to the data store. Therefore, events have less impact on the thread that is issuing the event because the thread does not have to wait for the relatively slow writing out of the event. This is one way in which Datastreams tries to minimize the instrumentation effect.

The work presented in this thesis uses data collected using DSKI and DSUI to verify the correctness of system components and to evaluate the overhead of those components.

3.2 Computation Component Set Manager

Under standard Linux, components are identified in a wide range of ways that do not necessarily have anything to do with specifying computation level control, scheduling semantics, nor measuring the performance of computations. For example, a set of process identifiers and thread identifiers are maintained by the system but the identifiers given are different under every execution of a given application. This presents a challenge both for measurement and for computation control because the precise control and measurement that we want needs a namespace that can be used to refer to different threads. Most important is that our way of describing measurement and scheduler configuration must use names that can be used effectively across all invocations of an application.

The Computation Component Set Manager allows an application component to declare itself to the system using an identifier which is the same across all instances of that application. The application component can then be referred to by outside subsystems. In addition, CCSM allows components to be grouped into sets that can also be referred to across instances of an application.

3.3 Proxy Management

Concurrency control semantics are an important part of the behavior of a system. If the concurrency control semantics are not coordinated with the scheduling layer semantics then the scheduling layer semantics may be compromised or overridden. For every set of scheduling semantics, it is difficult and impractical to implement an independent concurrency control layer. A configurable concurrency control layer reduces the effort to implement customized concurrency control semantics. The approach presented here permits the generalized tracking of the tasks awaiting a mutex and the optional customization of the selection semantics for the next owner when a mutex is released.

Proxy Management (PMGT) is a configurable concurrency control layer that serves as the essential core of the integration of concurrency control and scheduling semantics. It can be viewed as achieving the same goal as Priority Inheritance while generalizing the approach to make as few assumptions as possible about the scheduling semantics under which the set of threads using the concurrency control are managed. The essential task of Proxy Management is, like Priority Inheritance, to minimize the time during which a task holding a mutex can block the progress of a task the system would prefer to run.

PMGT was started by Noah Watkins and contributed to by Jared Straub but its completion was part of the work presented in this thesis [20]. In the implementation at the start (IAS) of the work described in this thesis, the proxy for a task was tracked by Proxy Management but several aspects of generalization remained. Additions and refinements to generalizing the Proxy Management semantics presented as part of this thesis fall into several categories. First, the ability to configure the semantics of the selection of the next owner of a mutex from the set of tasks waiting for it was added. Second, the relationship of PMGT to scheduling was generalized by creating a set of function calls that can be instantiated for any scheduling layer. Third, the accuracy of the information provided to the scheduling layer was improved. Fourth, support for deadlock detection was added to the Proxy Management algorithms and data structures.

This section first presents the basic design of PMGT and then gives an overview of the algorithms used by PMGT.

PMGT implements concurrency control that is not dependent on any particular scheduling semantics by recording a graph representation of the relationships among mutexes, the tasks that own the mutexes, and the tasks that are awaiting each mutex. We call this graph representation an m-blocking chain. All but one of the tasks in an m-blocking chain are awaiting a mutex under normal conditions. One task, the proxy, will not be awaiting a mutex. All of the other tasks in the chain are waiters because they are awaiting a mutex and they are blocked due to the proxy task. The proxy task must release one of the mutexes it holds for any other tasks in the m-blocking chain to make progress.

It is important for the scheduling layer to know who the proxy is for any task because the proxy determines what task should be run for a given task to eventually make progress. Therefore, if an important task is waiting on a mutex then its proxy should be scheduled. However, it is important to note that the proxy may not be runnable because proxies can block for other standard reasons under the process model. Thus, it is not always possible to reduce the blocking time of an important task by running the proxy.

The purpose of both PI and PMGT is to identify a task which can be run to minimize the blocking time of more important tasks. PI does this directly by using the priority of the tasks involved, essentially causing the owner of a mutex to inherit the scheduling criteria of its most important waiter. In effect, under priority scheduling this ensures that the task owning the mutex will be chosen whenever its most important waiter would be chosen to run. In contrast, PMGT directly tracks the proxy of each waiting task and permits the scheduler to choose the task it wishes to run directly and then to run its proxy instead if appropriate. PI and PMGT thus make the same choices under priority scheduling semantics but do so using different mechanisms. In addition, the PMGT semantics work under an extremely wide range of scheduling semantics since PMGT essentially ignores scheduling criteria.

The semantic generality of Proxy Management does come at a cost. PI does not change

the method by which the scheduler chooses the next thread to run as much as PMGT does. The reason for this is part of why PMGT is more general than PI. In effect, each task that has a proxy can be chosen by the scheduler but the proxy may or may not be runnable at that moment. Thus, the scheduler may have to check more than one candidate when choosing the next task to run.

Under this model, the overhead added to a scheduling decision by PMGT depends on the time required to examine the proxy for a task and the number of times that a proxy is examined. Therefore, the overhead is a function of the number of tasks m-blocked on the system because the proxy for some subset of these tasks will be considered during a scheduling decision. In theory, the PMGT overhead of a scheduling decision is unbounded. However, in practice, the number of m-blocked tasks and the number of those whose proxies are not runnable is likely to be bounded. Further, the current approach could be optimized by making an m-blocked task selectable only if its proxy is runnable.

The m-blocking chains recorded by PMGT can be described using a small set of definitions. These definitions can also be used to express how the m-blocking chains change as a result of mutex operations.

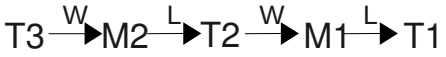


Figure 3.1. A Simple Blocking Chain

Figure 3.1 depicts a simple m-blocking chain. Tasks are prefixed with T and mutexes are prefixed with M. In this chain, an arrow denotes a relation. An L above an arrow indicates that a task owns a mutex and a W indicates that a task is awaiting a mutex.

To describe an m-blocking chain, it will be useful to be able to discuss the sets of tasks and mutexes on the system. We denote the set of tasks in the system ST, as shown in Definition 3.1. We denote the set of mutexes on the system as SM, as shown in Definition 3.2.

Definition 3.1 Set of Tasks on the System

$$ST = \{t \mid t \text{ is a task on the system}\}$$

Definition 3.2 Set of Mutexes on the System

$$SM = \{m \mid m \text{ is a mutex on the system}\}$$

We denote the ownership of mutex m by a task o as $Owns(o, m)$ and we denote that a task w is directly awaiting a mutex m as $DAwaits(w, m)$. A task w is directly awaiting a mutex m when w has made a request to lock m but has been blocked because some other task has already locked m. The ownership relation is defined in Definition 3.3 and the direct waiting relation is defined in Definition 3.4.

Definition 3.3 Ownership Relation

$$\begin{aligned} Owns(o, m) \quad = \quad & \{(o, m) \mid o \in ST \wedge m \in SM \wedge \\ & \text{o is the owner of m}\} \end{aligned}$$

In Figure 3.1, there are two "Owner" relations: $Owns(T1, M1)$ and $Owns(T2, M2)$. Also, in the figure, there are two "DAwaits" relations: $DAwaits(T2, M1)$ and $DAwaits(T3, M2)$.

There are three other relations that are not depicted in Figure 3.1 by arrows because these relations are derived from the $Owns$ and $DAwaits$ relations. The first derived relation is the indirect waiting relation. A task w that is directly awaiting a mutex $m2$ with owner o is also indirectly awaiting a mutex m if o is directly awaiting on m. Additionally, any task that is indirectly awaiting $m2$ is also indirectly waiting m. We define the indirect waiting relation in Definition 3.5.

In Figure 3.1, $T3$ is indirectly awaiting $M1$. Often, distinguishing between directly and indirectly waiting is unnecessary and we simply say that a task t is awaiting a mutex m. We define the $Awaits$ relation in Definition 3.6.

Definition 3.4 Direct Waiting Relation

$$\begin{aligned} DAwaits(w,m) &= \{(w,m) \mid w \in ST \wedge m \in SM \wedge \\ &\qquad \text{w is blocked in a mutex lock operation on m}\} \end{aligned}$$

Definition 3.5 Indirect Waiting Relation

$$\begin{aligned} IAwaits(w,m) &= \{(w,m) \mid \exists\, m2, o\, (w, o \in ST \wedge m, m2 \in SM \wedge \\ &\qquad Owns(o,m2) \wedge DAwaits(o,m) \wedge \neg Owns(w,m) \wedge \\ &\qquad (DAwaits(w,m2) \vee IAwaits(w,m2)))\} \end{aligned}$$

In Figure 3.1, for example, task $T3$ has the waiting relation $Awaits(T3, M2)$ because of a direct waiting relation and $Awaits(T3, M1)$ because of an indirect relation.

The second derived relation is the "m-blocked" relation. Most operating systems use a "blocked" state to indicate when a task is not runnable. PMGT effectively creates a new "m-blocked" state that indicates when a task is awaiting a mutex. Consequently, the blocked state now refers to any other reason for a task not being runnable. A task w awaiting a mutex m is m-blocked on the owner of m, task o, because o is one of the reasons because of which t cannot run. We define the m-blocked relation in Definition 3.7.

In Figure 3.1, there are thus two m-blocking relations for task $T3$: $MBlocked(T3, T2)$ and $MBlocked(T3, T1)$. If a task A is awaiting a mutex M then A is blocked on the owner of M.

The proxy relation is the final derived relation. For a task t and a task u, we assert that t is the proxy for u when u is m-blocked on t and t is not m-blocked, as shown in Definition

Definition 3.6 Awaits Relation

$$\begin{aligned} Awaits(w,m) &= \{(w,m) \mid w \in ST \wedge m \in SM \wedge \\ &\qquad (DAwaits(w,m) \vee IAwaits(w,m))\} \end{aligned}$$

Definition 3.7 The MBlocked Relation

$$MBlocked(w, o) = \{(w, o) \mid \exists m\, (w, o \in ST \wedge m \in SM \wedge Owns(o, m) \wedge Awaits(w, m))\}$$

3.8. Task t is not awaiting a mutex and, if t is runnable, the scheduler can run t in place of u. T1 is the task in Figure 3.1 that satisfies Definition 3.8, and, thus, T1 is the proxy for all of the other tasks in the chain: $Proxy(T1, T2)$ and $Proxy(T1, T3)$.

In general, knowledge of the proxy relation is useful only when a proxy is runnable. If the proxy is blocked for any of a number of reasons then the proxy is not of interest to the scheduler. When the proxy task is runnable, the scheduler may pick a task that is m-blocked on the proxy and the proxy can be run in its place. By running the proxy, the system is potentially reducing the blocking time of the task.

Definition 3.8 The Proxy Relation

$$Proxy(t, u) = \{(t, u) \mid \forall v\, (t, u, v \in ST \wedge MBlocked(u, t) \wedge \neg MBlocked(t, v))\}$$

Mutex operations change the structure of an m-blocking chain and, thus, change which waiting, m-blocking, and proxy relations exist. PMGT tracks the set of waiting relations and the set of proxy relations on the system and modifies these sets when mutex operations occur. We denote the set of waiting relations as SWR and the set of proxy relations as SPR, as shown in Definition 3.9. Initially, these sets are empty because no mutexes are held by tasks on the system. The goal of PMGT is to represent these sets accurately at any given moment. When an operation on a mutex is performed, updates to the set contents are required and there are periods during execution when the representation is being updated. The set of m-blocking relations is not considered here because this set is not tracked by PMGT.

Definition 3.9 System Waiting and Proxy Relations

$$\begin{aligned} SWR &= \{(t, m) \mid t \in ST \wedge m \in SM \wedge Awaits(t, m)\} \\ SPR &= \{(t, u) \mid t, u \in ST \wedge Proxy(t, u)\} \end{aligned}$$

The next part of this section considers what happens to these sets during four different scenarios: (1) when a task m-blocks, (2) when a task aborts an attempt to lock a mutex, (3) when a task unlocks a mutex, and (4) when a task steals a mutex.

When A Task M-Blocks

A task $T1$ that tries to lock a mutex M, held by a task $T2$, will m-block on $T2$ until $T2$ releases the mutex. When $T1$ m-blocks on $T2$, it merges the m-blocking chain associated with $T1$ into the m-blocking chain associated with $T2$. New waiting and proxy relations must be created for the tasks in the m-blocking chain associated with $T1$ because these tasks are now related to the tasks in the m-blocking chain associated with $T2$.

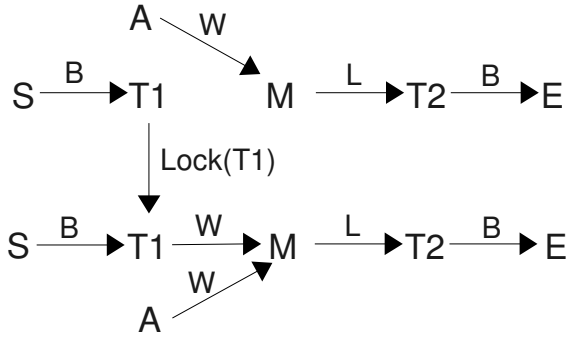


Figure 3.2. A Task M-Blocking

For example, Figure 3.2 depicts a task $T1$ trying to lock a mutex M that is owned by another task $T2$. In the figure, S represents the blocking chain associated with $T1$ at the start of the locking operation. The tasks in S are the set of tasks for which $T1$ is the proxy and the mutexes in S are the mutexes owned by any of the tasks in S or $T1$. We denote the

set of tasks in S as S_{task} and the set of mutexes in S as S_{mutex}. For the situation in Figure 3.2, S_{task} and S_{mutex} are:

$$\begin{aligned} S_{task} &= \{t \mid t \in ST \wedge MBlocked(t, T1)\} \\ S_{mutex} &= \{m \mid \exists t\, (t \in (S_{task} \cup \{T1\}) \wedge m \in SM \wedge Owns(t, m))\} \end{aligned}$$

E, in the figure, represents the portion of the blocking chain located after $T2$. The set E will be empty if $T2$ is not m-blocked. If E is not empty, the set of tasks in E, denoted E_{task}, contains the tasks on which $T2$ is m-blocked and the set of mutexes in E, denoted E_{mutex}, contains the mutexes that $T2$ awaits. E_{task} and E_{mutex} are:

$$\begin{aligned} E_{task} &= \{t \mid t \in ST \wedge MBlocked(T2, t)\} \\ E_{mutex} &= \{m \mid m \in SM \wedge Awaits(T2, m)\} \end{aligned}$$

A is the portion of the m-blocking chain that is awaiting M at the start of the locking operation. A_{task} contains the tasks awaiting M and which are thus m-blocked on $T2$ at the time $T1$ attempts to lock M. A_{mutex} contains all of the mutexes owned by the tasks in A_{task}. A_{task} and A_{mutex} are:

$$\begin{aligned} A_{task} &= \{t \mid t \in ST \wedge Awaits(t, M) \wedge \neg MBlocked(t, T1) \wedge t \neq T1\} \\ A_{mutex} &= \{m \mid \exists t\, (t \in A_{task} \wedge m \in SM \wedge Owns(t, m))\} \end{aligned}$$

Figure 3.2 thus depicts the m-blocking chain associated with $T1$ merging into the m-blocking chain associated with $T2$ because of the lock operation performed by $T1$. PMGT creates new waiting and proxy relations to represent the merging of the two m-blocking chains. $T1$ moves from a runnable state to an m-blocked state once the chains have been merged. The set S is depicted with a "B arrow" between it and $T1$ to indicate that all of its tasks are m-blocked on $T1$. Likewise, there is a "B arrow" from $T2$ to E to indicate that

$T2$ is m-blocked on all tasks in E. The set A is depicted with a "W arrow" that points to M because the tasks in A are awaiting M.

After the lock operation, $T1$ is directly waiting on M and all of the tasks in S are indirectly waiting on M. Also, the tasks in S and $T1$ are indirectly awaiting any mutexes in E. The change in waiting relations as a result of $T1$ trying to lock M is described by:

$$\begin{aligned} SWR' &= SWR \cup \{(s, m) \mid s \in (S_{task} \cup \{T1\}) \wedge \\ &\quad m \in (\{M\} \cup E_{mutex})\} \end{aligned} \tag{3.1}$$

Initially, $T1$ is the proxy for every task in S. However, $T1$ is m-blocking and therefore can no longer be a proxy. If E is empty then $T2$ will be the new proxy for $T1$ and all of the tasks in S. Otherwise, the proxy will be the task in E that is not m-blocked on any other task. Note that the waiting and proxy relations for A_{task} are unaffected by $T1$ m-blocking on M. The update of the proxy is described by:

$$\begin{aligned} SPR' &= (SPR - \{(T1, s) \mid s \in S_{task}\}) \cup \\ &\quad \{(e, s) \mid e \in (\{T2\} \cup E_{task}) \wedge \\ &\quad s \in (S_{task} \cup \{T1\}) \wedge Proxy(e, s)\} \end{aligned} \tag{3.2}$$

When A Task Aborts an Attempt to Lock a Mutex

When a task $T1$ aborts an attempt to lock a mutex M then the m-blocking chain associated with $T1$ is split into two blocking chains. One chain is associated with $T1$ and the other chain is associated with the owner of M, $T2$. Some waiting and proxy relations must be removed from the tasks in the m-blocking chain associated with $T1$ because the m-blocking chain associated with $T2$ is no longer part of the same chain. Additionally, $T1$ becomes the proxy for all of the tasks in the blocking chain associated with it. The abort operation is essentially the locking operation in reverse.

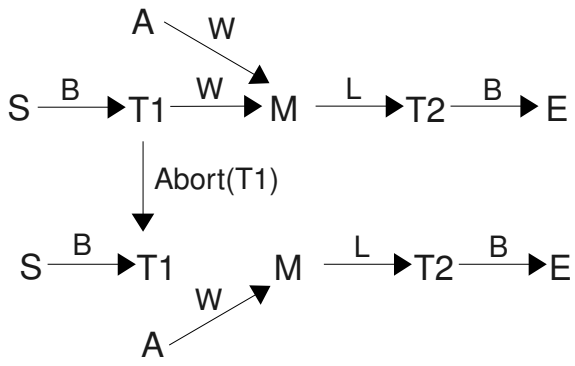


Figure 3.3. A Task Aborting an Attempt to Lock a Mutex

Figure 3.3 depicts a task $T1$ aborting an attempt to lock mutex M which is owned by task $T2$. This figure uses S to represent the portion of the m-blocking chain that contains tasks m-blocked on $T1$. A represents the portion of the blocking chain that contains tasks, other than those in S and $T1$, that are awaiting M. E is the portion of the m-blocking chain containing tasks on which $T2$ is m-blocked. The waiting and proxy relations for the tasks in A do not change because these tasks have the same relations to the tasks in E at the end of the abort.

When $T1$ aborts, it becomes runnable and it no longer has any waiting or proxy relations. Additionally, the tasks in S are no longer waiting on M or any of the mutexes in E. The change in waiting relations is described by:

$$\begin{aligned} SWR' &= SWR - \{(s,m) \mid s \in (S_{task} \cup \{T1\}) \wedge \\ &\quad m \in (\{M\} \cup E_{mutex})\} \end{aligned} \tag{3.3}$$

The set of proxy relations is updated as described in Equation 3.4. The relations for S_{task} and $T1$ are removed and new relations with $T1$ as the proxy are added.

$$\begin{aligned} SPR' &= (SPR - \{(t,s) \mid t \in ST \wedge s \in (S_{task} \cup \{T1\})\}) \cup \\ &\quad \{(T1,s) \mid s \in S_{task}\} \end{aligned} \tag{3.4}$$

When A Task Unlocks a Mutex

A task $T2$ that is releasing a mutex M is leaving the m-blocking chain associated with $T2$. $T2$ is initially the proxy for all of the tasks in this m-blocking chain because $T2$ is not m-blocked. During the release operation, the best waiter on the mutex must be selected to receive pending ownership of M and this task is the new proxy for the chain. PMGT has no knowledge about how to select the best waiter because the best waiter is determined using the scheduling semantics of the set of tasks awaiting M and PMGT does not assume any particular scheduling semantics. Therefore, we define a "Best" function which must be specified to PMGT to enable PMGT to determine the best task among a set of tasks using the scheduling semantics associated with the tasks. This function is given in 3.10.

Definition 3.10 The Best Function

$$Best(\{T\}) \quad = \quad \text{the best task where } t \in T$$

In this case, the $Best$ function is used to choose among the tasks awaiting M by evaluating $Best(\{t \mid t \in ST \wedge DAwaits(t, M)\})$. PMGT updates the SWR and SPR during the release operation to reflect the implications of $T2$ releasing M.

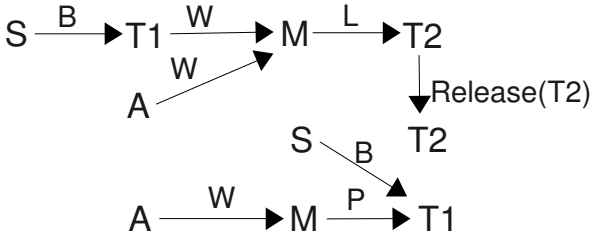


Figure 3.4. A Task Unlocking a Mutex

Figure 3.4 depicts a release operation. In this figure, task $T2$ is performing the unlock operation. When task $T2$ releases mutex M, the best direct waiter as determined by the $Best$ function, $T1$, is granted pending ownership of the mutex. When this occurs, $T1$ becomes

runnable and it is no longer m-blocked on $T2$. However, under the Linux mutex model, $T1$ is a "pending owner" because the mutex is not officially owned by $T1$ until $T1$ runs and takes ownership. During the period before $T1$ takes ownership, a better task is allowed to steal the mutex. If M is stolen then $T1$ will become m-blocked on the stealing task when it runs to try and make its pending ownership official. This figure uses S to represent the portion of the m-blocking chain that contains tasks m-blocked on $T1$ and A to represent the portion of the blocking chain that contains tasks, other than those in S and $T1$, that are awaiting N.

At the end of the release operation, task $T1$ is no longer awaiting any mutexes when it becomes the pending owner of M and, thus, its waiting relations are removed. Additionally, recall that the set S contains tasks that are blocked on $T1$. When $T1$ is granted pending ownership of M, the tasks in S are no longer awaiting M. Note that the set of waiting relations for the tasks in A does not change because these tasks are unrelated to $T1$ at the start of the unlock operation. The change in waiting relations is described by:

$$SWR' = SWR - \{(s, M) \mid s \in (S \cup \{T1\})\} \tag{3.5}$$

At the end of the unlock operation, task $T1$, the pending owner of mutex M, is the proxy for every task in A_{task} and S_{task} because $T1$ is not m-blocked. The change in proxy relations is described by:

$$\begin{aligned} SPR' = {} & (SPR - \{(T2, t) \mid t \in (S_{task} \cup A_{task} \cup \{T1\})\}) \cup \\ & \{(T1, s) \mid s \in (S_{task} \cup A_{task})\} \end{aligned} \tag{3.6}$$

When A Task Steals a Mutex

A task, $T1$, may steal a mutex, M, from a pending owner, $T2$, if $T1$ is better than $T2$. The *Best* function associated with the is used to determine if $T1$ is better than $T2$ by

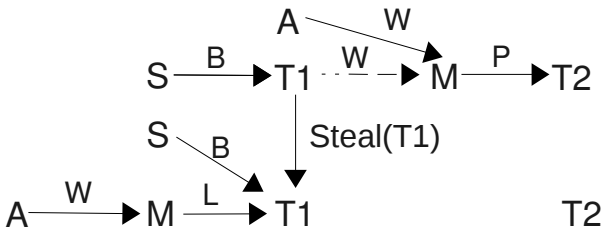


Figure 3.5. A Task Stealing a Mutex

checking if $T1 = Best(\{T1, T2\})$. Figure 3.5 depicts a successful stealing scenario where the stealing task is $T1$, the mutex is M, and the pending owner is $T2$. S and A in the figure represent portions of the m-blocking chain whose waiting and proxy relations may change as a result of the stealing operation. When M is stolen, the same steps used to update the SWR and SPR sets for the mutex unlock operation are applied with $T1$ receiving M and $T2$ releasing it. When $T2$ eventually runs, it will discover that it is no longer the pending owner of M and it will become one of M's waiters and thus have $T1$ as its proxy.

In the figure, a dotted "W arrow" indicates that $T1$ could be awaiting M when it tries to steal M. This circumstance arises when a waiting task is unexpectedly awaken and tries to steal the mutex. In this case, some extra accounting is necessary if $T1$ manages to successfully steal M because the SWR and SPR sets must be updated to reflect that $T1$ is no longer a waiter by removing any relations related to its previous status as a waiter. This change is described by:

$$SWR' = SWR - \{(T1, M)\} \tag{3.7}$$

$$SPR' = SPR - \{(T2, T1)\} \tag{3.8}$$

PMGT provides configurable concurrency control by tracking waiting and proxy relations. The tracked waiting and proxy relations are updated during the blocking, abort, release, and

stealing scenarios to reflect the structure of the resulting m-blocking chains.

3.4 Hierarchical Group Scheduling

On a general purpose system, applications with quite different semantics can be present. Virtually all conventional operating systems provide a programming model with a single semantics. Specifically, priority driven preemptive scheduling which controls each thread individually. Some particularly advanced or special purpose system may provide deadline scheduling or minor variations on priority scheduling.

Unfortunately, developers often find it difficult to map the desired application semantics on to the priority based semantics the system makes available. This difficulty often arises because priorities are simply not appropriate but it can also arise because controlling each thread on the system individually prevents the collaborative semantics a multithreaded application may desire. In response to this gap, developers often use concurrency control to achieve specific application behavior effects. However, in a system which permitted developers to directly implement the application behavior they desired in the form of a customized scheduler, no such use of concurrency control for scheduling effects would be necessary.

Hierarchical Group Scheduling (HGS) resolves this conflict by providing the developer with the ability to flexibly configure and precisely control behavior at all levels from the individual application to the system as a whole. HGS does this by permitting developers to (1) choose among a range of standard schedulers, by permitting developers to (2) write a custom scheduler, by permitting developers to (3) create scheduling hierarchies to describe how different aspects of the system are controlled and by (4) integrating scheduling with concurrency control.

The rest of this section will provide an overview of the set of standard schedulers, the framework in which custom schedulers can be written, and how the scheduling hierarchy for the system is specified to provide context that helps in understanding the work presented in this thesis. This thesis focuses on completing the work done by previous students on the

integration of scheduling and concurrency control which had many important capabilities but also several important deficits. The completion and testing of this work is discussed in greater detail in Section 3.5 and Section 3.8, respectively.

There are several familiar schedulers that have already been written or should be relatively easy to write, including: static priority, dynamic priority, round robin, and earliest deadline first. However, the set of schedulers on the system is not fixed. HGS provides a framework that allows developers to write a custom scheduler and configure the system to use the scheduler with relative ease. The methods for writing custom schedulers are discussed in the documentation for HGS [15]. One example of a custom scheduler that has already been written controls multiple pipeline computations in a way that ensures balanced progress among the pipelines [1]. Additionally, as part of testing the completed integration of scheduling and concurrency control, another example of a custom scheduler is presented in Section 3.8.

HGS uses a hierarchy consisting of tasks and groups to specify system behavior. Both tasks and groups can be a member of one or more groups. A scheduler is associated with each group and chooses among group members when invoked. A given scheduler can be associated with more than one group but since each group's member data is used separately, so this is best viewed as multiple instances of the same algorithm being associated with different groups. The hierarchical design is discussed more fully in [20].

At scheduling time, the scheduler associated with the top group of the HGS hierarchy is invoked. It may choose to query a group which is one of its members. This query takes the form of a nested call to the subordinate group's scheduler. At the level of a given group several queries to subordinate groups may be required. As a result, the pattern of execution within the scheduling hierarchy is one of a recursive search through a decision tree until a task is picked.

HGS tries to impose few restrictions on scheduler semantics in order to allow a wide range of schedulers to be implemented. In support of this goal, HGS relies on PMGT to

permit developers to configure concurrency control semantics that match the scheduling semantics implemented by the HGS scheduling hierarchy. The precision of computation behavior control is maximized when the semantics of the schedulers and concurrency control affecting a set of threads match one another.

In Section 3.7.2, this work presents a new approach to the integration of HGS and PMGT which provides support for multiprocessor systems and allows schedulers to explicitly specify concurrency control semantics.

3.5 Proxy Management Extensions

The specification of the concurrency control semantics for a system consists of two parts: the mutex policy associated with each mutex and the proxy policy of the system. A mutex policy defines how the tasks that are directly awaiting the release of a mutex are handled, including: which waiter is selected to receive ownership of the mutex when it is released by the owner and when a task is allowed to steal the mutex. A proxy policy determines how the tasks that are m-blocked on a proxy influence the scheduling criteria of the proxy. To be effective as a configurable concurrency control layer, PMGT must allow both mutex and proxy policies to be configured.

In the IAS, PMGT implemented the generalized algorithms necessary to support the configuration of mutex and proxy policies but it did not actually allow these policies to be configured. Instead, in the IAS, PMGT assumed a basic FIFO mutex policy for all mutexes and relied on HGS exclusively to provide the proxy policy for the system.

A set of callbacks now exists to allow different mutex policies to be defined and each mutex can be associated with a different mutex policy. FIFO and priority based mutex policies have been implemented using these callbacks. A second set of callbacks exists to allow the scheduling layer to specify the proxy policy for the system. These callbacks have been used to implement a PI proxy policy for the PREEMPT-RT scheduling layer and to implement an SMP aware proxy policy for HGS. This section further discusses the details of

the mutex and proxy policy callbacks.

3.5.1 Mutex Policy Configuration

To fully specify system concurrency control semantics, a policy must be specified to determine how the tasks directly awaiting the release of a mutex are handled. PMGT addresses this issue by providing a set of callbacks that allow developers to implement a wide range of mutex policies. Additionally, PMGT allows an individually customized mutex policy to be specified for one or more mutexes.

PMGT assumes little about the concurrency control semantics of the system in order to allow a wide range of concurrency control semantics to be configured. As a result, there are some decisions that PMGT cannot make. For example, when the owner of a mutex releases the mutex, the best waiter on the mutex receives pending owner status for the mutex. PMGT has no way to know what criteria should be applied to the waiters to determine which is the best. Similarly, if a task attempts to steal a mutex, PMGT does not know if the stealing task should be allowed to steal the mutex.

Choosing the best waiter when a mutex is released obviously interacts with the scheduling semantics because the waiter that the system scheduling semantics would most like to run, if it were runnable, ought to be the one that is given the mutex and would thus become runnable. Additionally, the scheduling semantics determine if a task is allowed to steal a mutex because the scheduling semantics must prefer the stealing task to the pending owner.

What component should make a decision when PMGT cannot? In the case where all tasks are managed under a uniform system scheduling semantics the scheduling algorithm involved is sufficient to choose the best waiter. For example, if priority semantics are being used by the system to schedule tasks then the best waiter on a mutex is the task with the best priority. However, we do not wish to assume that all system configurations will involve uniform scheduling semantics for all tasks. For example, under HGS different sets of tasks may be controlled under different scheduling semantics. In the case where the

system scheduling semantics is known at build time the system developer can, in theory, write a routine for choosing the best waiter that can take into account the mix of scheduling semantics under which the various waiting tasks may be controlled. This can obviously be a very difficult problem to solve but it is the responsibility of the developer configuring the system to be able to solve it. Otherwise, the developer's desired system semantics are not feasible.

A further complication is that HGS permits schedulers to be developed as part of applications and thus to be loaded dynamically. This implies that the mix of scheduling semantics represented by a set of waiters can not even be predicted at system compile time under all circumstances. Therefore, in creating this framework for the integration of scheduling and concurrency control, we do not attempt to solve every possible problem. Instead, we allow a developer to supply a set of callbacks called a mutex policy that is associated with one or more mutexes.

Different sets of mutexes in the system may be used by different sets of tasks. The set of mutexes used to implement basic Linux services can be used by any task on the system. However, mutexes used within a specialized module might be used by a set of tasks restricted to a single application.

Our system currently only addresses OS level mutexes but extension to user level mutexes is a fairly obvious next step. User level mutexes used by a single application could be attractive targets for specialized semantics including waiter selection. User-level PI-mutexes dynamically create OS level mutexes when the first waiter m-blocks, so the extension of PMGT to include PI-mutexes would be a relatively simple extension of the capabilities described in this thesis.

In the case of a mutex used by a kernel module or single application, we can imagine that the developer might like to provide a waiter selection algorithm with specialized semantics. Thus, the system supports configuring the waiter selection routine on a per-mutex basis. However, since specifying the waiter selection semantics for each mutex in the Linux OS

individually is completely infeasible the scheduling layer configured in the system can choose any routine as the default policy used for all mutexes unless overridden.

There are two mutex policy callbacks that determine mutex semantics: *top* and *can_steal*. The *top* callback is invoked to choose the best waiter to become the pending owner when the mutex is released. Conceptually, *top* is responsible for evaluating the *Best* function specified for the mutex. This callback is called *top* instead of *best* because PREEMPT-RT uses the word *top*. Obviously, FIFO and priority based algorithms will be popular choices and thus we provide them as conveniences. The *can_steal* callback decides if a mutex can be stolen.

Though sufficient to decide the best waiter, a policy that selects among an unordered set of waiters is likely to be computationally more expensive than a policy that orders the waiters and selects the waiters in order. To reduce overhead, facilities are provided to allow a mutex policy to order the waiters on a mutex.

The *mutex* data structure contains a union of data structures, called *waiters*, that can be used to order the mutex's waiters. The use of a union limits the available data structures to compile time configuration but this is not an issue because there are only a small number of data structures that are suitable for this task and currently there is no need for a mutex policy to specify a new data structure at runtime. However, if a developer wanted a new data structure then it could be added to the union.

There are four mutex policy callbacks that maintain the *waiters* union: *init*, *insert*, *remove* and *has_waiters*. The *init* callback initializes the data structure in the *waiters* union which should be used. *Insert* inserts waiters into the initialized data structure and *remove* deletes hem. Finally, *has_waiters* queries the initialized data structure to see if there are any waiters.

PMGT provides FIFO and priority mutex policies that demonstrate the use of these callbacks because FIFO serves as a simple example and priority is the most commonly used form of scheduling semantics. The FIFO policy uses a linked list as a queue to order the waiters on the mutex. When the best waiter for the mutex is requested, the head of the

queue is removed. The priority policy uses a linked list that orders waiters by priority. It returns the waiter with the best priority when the best waiter is requested. This policy produces the priority semantics used by the PREEMPT-RT scheduling layer.

3.5.2 Proxy Policy Configuration

Although PMGT independently performs most of the accounting about proxy relations, the scheduling layer must also be aware of the proxy relations because the proxy relations affect the scheduling criteria for the threads involved in the relation. Proxy relations are thus information used by the scheduling layer to make a scheduling decision. PMGT allows developers to implement a set of callbacks, called a proxy policy, which determines how the scheduling layer handles proxy relations.

The algorithms used by PMGT do not make any assumptions about how the scheduling layer handles proxy relations. However, in the IAS, PMGT was tightly bound to HGS and this coupling prevented PMGT from being used with other scheduling layers. The proxy policy callbacks were created to present generic notifications about proxy relations to the scheduling layer without assuming any particular scheduling layer is present in order for PMGT to be useful to many scheduling layers. These callbacks are invoked when proxy relations are created and destroyed.

In addition to being tightly bound to HGS, the IAS often incorrectly identified tasks as proxy tasks. In an m-blocking chain, every task in the interior of the chain, between an m-blocked task and its proxy, was temporarily identified as the proxy in the iterative process of finding the real proxy. The proxy policy callbacks eliminate these incorrect interim states by only notifying the scheduling layer when the real proxy is found.

We assume that the scheduling layer does not change at runtime because we are primarily interested in changing the set of schedulers on the system and not how those schedulers are managed. Therefore, the scheduling layer notified by PMGT can only be configured at kernel compile time. Additionally, kernel compile time configuration of the scheduling layer reduces

the overhead of PMGT by eliminating the need for additional concurrency control required in order to handle runtime changes to the scheduling layer configuration.

Ideally, a scheduling layer would not need to make any modifications to PMGT data structures in order to react to proxy relation changes. This would provide a clean separation of PMGT and scheduling data structures. However, if the PMGT and scheduling data structures are completely separate then, when proxy policy callbacks occur, the scheduling layer may need to hold the concurrency control for both sets of data structures in order to react to a proxy relation and this would likely create a greater constraint on concurrency than necessary.

In an effort to constrain concurrency less, PMGT allows the scheduling layer to store arbitrary data about waiters and proxies that is accessed using PMGT's internal concurrency control. Essentially, a scheduling layer is able to copy information that it would like to use during mutex operations into designated sections of PMGT data structures and then the scheduling layer can access this data while holding only PMGT concurrency control during mutex operations when notifications about proxy relations occur. The waiter and proxy data is opaque to PMGT and chosen as appropriate to the scheduling semantics. This approach reduces the number of data structures that must be accessed during mutex operations but when scheduling data structures are changed the data stored in the PMGT data structures may need to be updated to reflect those changes. Therefore, this trade-off is generally beneficial when the scheduling data needed during mutex operations changes much less frequently than mutex operations occur.

There are several proxy policy callbacks that the scheduling layer may implement to maintain data about waiters and proxies. A *task_init* callback can be specified to initialize data maintained about a proxy. Additionally, the scheduling layer can specify a *waiter_init* callback to initialize data maintained for each waiter. The structures available to these routines are specified by the scheduling layer by defining the *PMGT_SCHED_PROXY* and *PMGT_SCHED_WAITER* preprocessor constants, respectively. The *waiter_update* callback

can be provided by the scheduling layer to update the data stored on a waiter and its proxy when scheduling layer data changes. When scheduling layer data is updated, the scheduling layer is responsible for executing *pmgt_update* which invokes *waiter_update* with the correct concurrency control to access the waiter and proxy data structures held.

There are four callbacks that can be implemented by the scheduling layer to receive notifications about changes to the system's set of proxy relations: *waiter_move_prepare*, *waiter_move*, *task_finalize*, and *waiter_destroy*. An invocation of *waiter_move_prepare* indicates that PMGT is searching for a new proxy for a waiter. This callback refers to movement because waiters are associated with a proxy and, when the proxy for a waiter changes, we say the waiter is moved from the old proxy to the new proxy. The *waiter_move* callback indicates a new proxy for a waiter has been discovered. Finally, *task_finalize* is an optimization that allows scheduling criteria of a proxy to be modified after several waiters have been associated with that proxy, through multiple invocations of *waiter_move*, instead of these modifications being performed for each invocation of *waiter_move*. *Waiter_destroy* tells the scheduling layer that a task is no longer a waiter.

Section 3.7.2 presents an example of how to implement a proxy policy, for HGS, to handle proxy relations between many different schedulers. This example is important because it demonstrates how a scheduling layer can mediate between multiple schedulers without knowing the scheduling algorithm of each scheduler. Additionally, Section 3.6 presents an example of how to create a proxy policy which implements Priority Inheritance.

3.5.3 Deadlock Detection

Deadlock is a common pitfall in concurrent software and a concurrency control implementation that is unprepared for deadlock to occur is a risk to the stability of the system. This section presents the deadlock detection and handling in PMGT.

The PREEMPT-RT patch has two approaches to deadlock. One detects it and then relies on a signal reception or a timeout among one or more of the deadlocked tasks to

resolve it. The other detects it during the mutex lock operation that would complete the cycle and aborts the lock operation with an error return. The PMGT approach to deadlock provides the same two choices. The basic approach of PMGT is quite similar to that of PREEMPT-RT but since the PMGT data structures are somewhat more complex there are some differences and additional subtleties.

If the proxy of an m-blocking chain tries to acquire a mutex that is owned by a task that is m-blocked on the proxy then a cycle will be created and, thus, deadlock occurs. In this scenario, the task that was the proxy can no longer be a proxy because it is m-blocked. Therefore, every task in the chain is m-blocked and no task is the proxy. Effectively, every task in a cycle is awaiting every mutex involved in the cycle. PMGT does not record that a task is awaiting a mutex it owns because representing this relation requires extra memory and this relation is not necessary for the deadlock detection used in PMGT. Instead, these relations are deduced from other relations.

Definition 3.11 A Task In a Cycle Involving a Mutex

$$\begin{aligned} InCycle(t, m) \quad = \quad & \exists o(t, o \in ST \wedge m \in SM \wedge \\ & Owns(o, m) \wedge Awaits(t, m) \wedge \\ & MBlocked(o, t)) \end{aligned}$$

Often it is useful to know if a task t is part of a cycle as a result of awaiting a mutex m. If the owner of m, task o, is m-blocked on t then a cycle exists because o is also m-blocked on itself. Definition 3.11 describes when a task t is part of a cycle involving a mutex m.

Additionally, a task may enter a deadlock if the task tries to acquire a mutex that is part of a cycle that already exists. Definition 3.12 states that a task t is connected to a cycle involving a mutex m when it is awaiting m and there is some task u that is in a cycle involving m.

PMGT represents the waiting relations for all m-blocked tasks and when deadlock occurs it must ensure that the appropriate waiting relations have been created for each task involved

Definition 3.12 A Task Connected to a Cycle Involving a Mutex

$$\begin{aligned} ConnectedCycle(t, m) \quad = \quad & \exists u \, (t, u \in ST \wedge m \in SM \wedge \\ & Awaits(t, m) \wedge \neg InCycle(t, m) \wedge \\ & InCycle(u, m)) \end{aligned}$$

with a cycle. Recall that every task in a cycle is awaiting every mutex in the cycle and every task connected to a cycle is also awaiting every mutex in the cycle. This assertion is described by Equation 3.9.

$$\begin{aligned} SWR \quad \supseteq \quad & \{(t, m) \mid t \in ST \wedge m \in SM \wedge \\ & (InCycle(t, m) \vee ConnectedCycle(t, m))\} \end{aligned} \tag{3.9}$$

PMGT performs deadlock detection when searching an m-blocking chain to find a proxy for a set of tasks, NP. When a task m-blocks, NP includes the m-blocking task and any tasks for which that task is a proxy. At each link in the chain, PMGT performs deadlock detection for every task T where $T \in NP$. Deadlock is detected if: (1) T is the same as the next task in the chain or (2) T is directly awaiting the next mutex in the chain. These conditions check if a task is in a cycle and if is connected to a cycle, respectively.

However, PMGT cannot always immediately stop when deadlock is detected. If NP is not empty then Equation 3.9 must still be satisfied because waiting relations still need to be recorded for some tasks. The task T for which deadlock was detected is removed from NP and the search continues until NP is empty. The search terminates when NP becomes empty because there is no work left to do and Equation 3.9 is satisfied because waiting relations have been created at each link in the cyclic chain for all of the tasks in NP.

At the end of the search, none of the tasks in a cycle is a proxy because all of the tasks are awaiting a mutex. Therefore, if deadlock is detected, no proxy is reported to the scheduling layer for any of the tasks involved in the deadlock. The tasks are marked as "searching for

a proxy" until the deadlock is resolved.

Figure 3.6 depicts the tasks, mutexes, and waiting relations of a deadlock scenario. Tasks $T1$, $T2$, and $T3$ are in a cycle. Task $T4$ is connected to the cycle. In the figure, $N_{x,y}$ denotes a record of a waiting relation between a task x and a mutex y. Note that every task is awaiting every mutex. Additionally, in this figure, W_z is placed at certain locations of the chain to indicate where PMGT stopped searching for a new proxy for a task z. All of the tasks are in the searching state because there is no proxy.

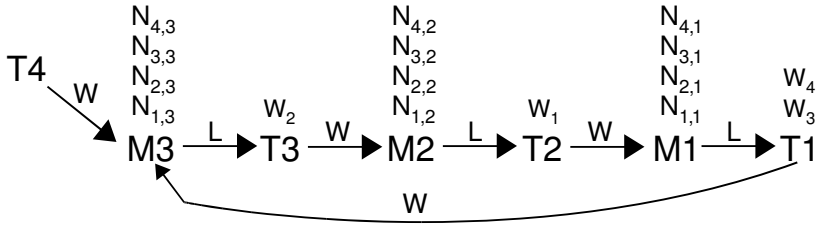


Figure 3.6. A Deadlock Scenario

In a deadlock scenario, deadlock handling starts after the appropriate waiting relations created by the deadlock have been recorded. Under Linux, users of the mutex API determine if a locking attempt should abort due to a deadlock by specifying options in the API. If the deadlock abort option is used then PMGT returns an error value when deadlock is detected. If the deadlock is permitted to occur then only reception of a signal or timeout of a participating task will resolve it. If a task in the cycle aborts a locking operation then that task will become the proxy for every other task in the m-blocking chain that results from breaking a link in the cycle.

3.5.4 PMGT Algorithms Supporting Mutexes

This section discusses the algorithms used by PMGT to implement a concurrency control layer with configurable semantics. Five algorithms are presented: (1) the task_blocks_on_rt_mutex algorithm, (2) the wakeup_next_waiter algorithm, (3) the try_to_steal algorithm, (4) the remove_waiter algorithm, (5), and the move_waiters algorithm. PMGT combines these al-

gorithms to implement support for the four scenarios presented in Section 3.3: (1) a task m-blocking, (2) a task releasing a mutex, (3) a task stealing a mutex, and (4) a task aborting a mutex lock operation. PMGT modifies the PREEMPT-RT mutex implementation in order to leverage some of the already established code and reduce the size of the PMGT patch. All of the algorithms presented here are called from within the PREEMPT-RT mutex implementation.

The PMGT algorithms use non-preemptable spin-locks for all concurrency control. The "lock" and "unlock" operations in these algorithms refer to non-preemptable spin-locks. They do not represent mutex operations. For each algorithm, the calling context holds the *mutex.lock* spin-lock that controls access to the data structures of the mutex that record the owner of the mutex.

Task_blocks_on_rt_mutex

The PREEMPT-RT mutex implementation calls the *task_blocks_on_rt_mutex* function when a task is m-blocking as a result of trying to acquire a mutex that is owned by another task. Figure 3.7 illustrates an example of this situation where $T1$ is trying to acquire M and is m-blocking on $T2$, the owner of M. PMGT provides a replacement for the PREEMPT-RT version of *task_blocks_on_rt_mutex* that sets up PMGT data structures instead of performing PI operations. The goal of *task_blocks_on_rt_mutex* is to update SWR and SPR as specified by Equations 3.1 and 3.2. Program 3.1 presents pseudo-code describing the semantics of the PMGT version of *task_blocks_on_rt_mutex*.

The calling context for *task_blocks_on_rt_mutex* provides a waiter data structure used to represent the task while it is m-blocked. The waiter data structure is used by PMGT and by the scheduling layer in various situations.

At line 1, the *find_proxy* set is initialized to track waiters for which a new proxy needs to be found, which are those in $S_{task} \cup \{T1\}$ in the figure. In theory, the proxy for $T1$ could be instantly known by looking at the proxy for $T2$. However, before this proxy relation can

Program 3.1 Task_blocks_on_rt_mutex

```
   task_blocks_on_rt_mutex(task, waiter, mutex)

     deadlock := false

1    find_proxy := empty set

2    lock(task.pi_lock)

3    init_waiter(waiter, task, mutex)

4    mutex.insert(waiter)

5    set_add(find_proxy, waiter)

6    for each waiter in task.waiters do
       sched_move_prepare(waiter, task)
     end_for
     if task.waiters is not empty then
       sl.task_finalize(task)
     end_if

7    if prepare_waiters(task, mutex, find_proxy) then
       deadlock := true
     end_if

8    unlock(task.pi_lock)

9    if adjust_chain(task, mutex, find_proxy) then
       deadlock := true
     end_if

10    return deadlock

   end_func
```

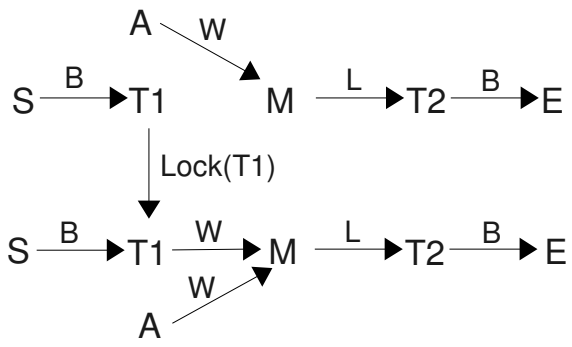


Figure 3.7. A Task M-Blocking

be recorded, PMGT must also update the SWR to reflect that $T1$ is blocking as specified by Equation 3.1 on page 35. This update adds waiting relations between $S_{task} \cup \{T1\}$ and $\{M\} \cup E_{mutex}$.

Line 2 acquires the *task.pi_lock* that protects the PMGT data structures associated with a task. The name of this lock indicates that it would be used for PI but this is not the case. Rather, this lock is re-used from the PREEMPT-RT mutex implementation to make PMGT a smaller patch. Holding the *task.pi_lock* allows the algorithm to initialize the waiter structure at line 3 to indicate that the task is directly awaiting the mutex. *Init_waiter* is described separately in Program 3.2. The *Init_waiter* function updates SWR according to Equation 3.1 by setting up the waiting relation between *task* and *mutex*.

Line 4 inserts the waiter structure in the data structure used to store the waiter structures for tasks that directly await the mutex (M). Note that the data structure and the insert routine can be configured per-mutex to support arbitrary semantics and the data structure tracking the waiters is opaque to PMGT. Line 5 adds the waiter ($T1$) to the *find_proxy* set because we have to find the proxy for the waiter.

The steps at 6 tell the scheduling layer, by calling *sched_move_prepare*, that a new proxy needs to be found for all of the tasks in S. The *sl.task_finalize* scheduling layer callback gives the scheduling layer a chance to modify the scheduling criteria of the old proxy, $T1$, after all of the waiters for the tasks in S have been marked as searching.

The steps at 7 call *prepare_waiters* to record the waiting relations that relate the tasks in S to *mutex* in order to to update SWR according to Equation 3.1. Also, *prepare_waiters* removes the proxy relations for the tasks in S and adds these tasks to the *find_proxy* set because a new proxy for these tasks must be found. *Prepare_waiters* also performs deadlock detection and we record that a deadlock has occurred if deadlock was detected. Note, however, that the deadlock algorithm performs deadlock detection on a per-waiter basis. When a deadlock is detected, PMGT may need to continue searching for a proxy on behalf of some waiters in order to record additional waiting relations to fully update SWR. In this case, we know that we need to find the proxy for the waiter created to represent $T1$ and, thus, we know the *find_proxy* set will not be empty.

Line 8 releases the *pi_lock* of $T1$ because the algorithm has finished modifying its PMGT data structures.

Adjust_chain is called by the steps at 9 to search for the new proxy for all of the waiters in the *find_proxy* set. *Adjust_chain* also performs deadlock detection and deadlock is recorded when it is detected in order to communicate deadlock to the calling context. The *adjust_chain* function completes the updates to SWR and SPR according to Equations 3.1 and 3.2. At line 10, *task_blocks_on_rt_mutex* informs the calling context if deadlock was detected. On receipt of a deadlock return value, the calling context will abort the locking operation if the blocking task has specified the deadlock abort option to the RT mutex API.

The *init_waiter* function in Program 3.2 is responsible for: initializing a waiter data structure, allowing the scheduling layer to initialize scheduling data stored in the waiter structure, and creating a representation of a waiting relation that records that a task is m-blocking due to trying to acquire a mutex. PMGT uses a *node* structure to record waiting relations. Normally, to access the waiter, the *waiter.lock* must be held. However, at this point, the *waiter* structure is not visible to any other tasks and, thus, it can be accessed without holding the *waiter.lock*. For the scenario in Figure 3.7, *init_waiter* is used to record the direct waiting relation $(T1, M)$.

Program 3.2 Init_waiter

```
   init_waiter(waiter, task, mutex)

1    task.waiter := waiter
     waiter.task := task
     waiter.mutex := mutex

2    new_node(waiter, NULL, mutex)

3    sl.init_waiter(waiter, task)
     waiter.state := PROXY_SEARCH

4    waiter.proxy_candidate := mutex.owner

   end_func
```

The steps at 1 in *init_waiter* perform basic initialization to give the task structure access to the waiter structure and the waiter structure access to the task structure. The mutex that *task* is m-blocking on is recorded. The direct waiting relation between *task* ($T1$) and *mutex* (M) is recorded at Line 2 by calling *new_node*. The arguments to this function are the *waiter*, the *via_task* and the *via_mutex*. The *via_mutex* field indicates the mutex involved in the waiting relation, M. For an indirect waiting relation, the *via_task* field indicates the waiter that was traversed immediately before reaching *via_mutex*. Here, *via_task* is NULL because this node represents a direct waiting relation.

The steps at 3 allow the scheduling layer to initialize the waiter data structure and record that a proxy has not yet been found for *waiter* by marking it as searching. In the IAS, the state of the waiter as reported to the scheduling layer was not tracked. As a result, many unnecessary calls were made to the scheduling layer during a single mutex operation and the unnecessary calls presented incorrect proxy information.

Line 4 sets the *proxy_candidate* for the waiter. The *proxy_candidate* is the task most recently examined during the search of an m-blocking chain. If the search has finished then *proxy_candidate* identifies the proxy.

Lines 1-8 of Program 3.1 determine the waiters for which a new proxy must be found and

Program 3.3 Adjust_chain

```
    adjust_chain(mutex, find_proxy)

1    deadlock := false
     start_mutex := mutex

2  move:
     task := mutex.owner
     lock(task.pi_lock)

3    set_union(task.waiters, find_proxy)

4    unlock(task.pi_lock)
     unlock(mutex.lock)
     # preemptable here
     lock(task.pi_lock)

5    if task.waiter is NULL then
       for each waiter in task.waiters do
         sched_move(waiter, task)
       end_for
       sl.task_finalize(task)
       goto out
     end_if

6    mutex := task.waiter.mutex
     lock(mutex.lock)

7    find_proxy := empty set

8    if prepare_waiters(task, mutex,
         find_proxy) then
       deadlock := true
     end_if

9    if find_proxy is empty then
       unlock(mutex.lock)
       goto out
     end_if

10   unlock(task.pi_lock)
     goto move

11 out:
     unlock(task.pi_lock)
     lock(start_mutex.lock)
     return deadlock

   end_func
```

line 9 calls the *adjust_chain* function, Program 3.3, to create waiting and proxy relations. For the scenario in Figure 3.7, *adjust_chain* creates the waiting and proxy relations between $S \cup \{T1\}$ and E that are specified by Equations 3.1 and 3.2.

In Program 3.3, the steps at 1 initialize a deadlock flag which records deadlock so that it can be communicated to the caller at the end of the function. Additionally, recall that *mutex.lock* is held by the calling context, *task_blocks_on_rt_mutex*. Thus, the steps at 1 store *mutex* in order to reacquire *mutex.lock* before exiting. In Figure 3.7 on page 54, *mutex* is M.

The steps at 2 indicate that the task that is the owner of *mutex* is the next location in the chain. The *pi_lock* for the owner of *mutex* is acquired to allow this function to modify its PMGT data structures.

Line 3 adds all of the waiters for which a new proxy needs to be found to the set of

waiters associated with the owner of the mutex. However, the owner of the mutex is not necessarily the proxy because it could itself be waiting for a different mutex. Therefore, no information is yet sent to the scheduling layer. In Figure 3.7, a new proxy needs to be found for all of the tasks in S.

PMGT does not know how long it will take to search for a proxy. Therefore, between the links of the blocking chain, PMGT enables preemption to allow more important work to take place on the CPU. The steps at 4 unlock all of the spin-locks that are held to enable preemption. After the preemptable section, *pi_lock* for the owner of *mutex* is again acquired because we may need to prepare waiters for the search to proceed to the next link in the chain. Note that a KUSP kernel configuration option exists to make the process of stepping down an m-blocking chain non-interruptible and non-preemptable. However, it is not clear which choice is best in different situations. Making the traversal of the m-blocking chain interruptible tends to reduce event response latency of the system because an interrupt can be serviced, which may cause a context switch, at each step of the traversal. This seems to obviously decrease the time an important task can be kept from running, and is why the interruptibility option is on by default. However, making the traversal non-interruptible tends to minimize the time during which a task's proxy is not known, and thus minimizes the time during which it is not possible to select the important process and thus run its proxy. This tends to increase the delay of important process which is blocking on a mutex. The two approaches both have an argument in their favor although the first seems dominant when considering behavior in the abstract. The configuration option is provided to permit developers to choose whichever is most important in their particular situation.

The steps at 5 check if the owner of the mutex is eligible as a proxy by checking if $task.waiter$ is $NULL$. The $task.waiter$ element is non-null if a task is m-blocked. If a proxy has been found then the scheduling layer is informed, by calling *sched_move* for each of the waiters in the $task.waiters$ set, that a proxy has been found for each of the waiters. Note that another task may have already informed the scheduling layer about some or all

of the waiters in the *task.waiters* set and, thus, *sched_move* does nothing if the scheduling layer has already been informed about a waiter. Additionally, if any waiters were moved, *task_finalize* is called to give the scheduling layer an additional chance to update scheduling criteria based on the full set of waiters. This is an optimization that allows most of the work to be performed after all of the waiters have been assigned instead of for each waiter. For instance, *task_finalize* performs priority adjustment under the PMGT based version of PI.

If the task at the current location in the chain is m-blocked then, in the steps at 6, we look at the waiter representing the task and retrieve the mutex on which the task is m-blocked. Thus, the search proceeds to the next mutex in the m-blocking chain.

At line 7, *find_proxy* is initialized to an empty set. PMGT may perform multiple searches simultaneously on behalf of different sets of tasks. Therefore, it is not possible to use the *find_proxy* set that was originally passed to *adjust_chain* because the proxy search for some or all of those waiters may have been performed by another search while $T1$ was preemptable. In Figure 3.7, *find_proxy* will always contain S because a single search is in progress. The steps at line 8 identify a new set of waiters for which a proxy needs to be found. If deadlock is detected by *prepare_waiters* then we record that a deadlock occurred.

The steps at 9 check to see if their are still waiters for whom a new proxy must be found. There may not be any waiters if another task took over searching on behalf of the waiters provided at the start of *adjust_chain* or the waiters were abandoned due to deadlock. If there are no more waiters for which we must search then the search is finished. The steps at 10 iterate the loop by returning to the *move* label to allow the search to proceed to the next link of the m-blocking chain. Essentially, this loop is searching through the tasks in E in Figure 3.7.

All points of exit from *adjust_chain* hold the *pi_lock* of the current task being examined for the search. The steps at 11 prepare to return to the calling context by dropping the *pi_lock* and then reacquiring the *start_mutex.lock* held by the calling context at the start of *adjust_chain*.

Ultimately, *adjust_chain* searches an m-blocking chain until a proxy is found or deadlock is detected. If a proxy is found then the tasks for which a proxy needs to be found are associated with the proxy and a success value is returned to Program 3.1. Otherwise, when deadlock is detected, a deadlock value is returned after SWR has been fully updated.

Program 3.4 Prepare_waiters

```
   prepare_waiters(task, mutex, find_proxy)

1    deadlock := false

2    for each waiter in task.waiters do
       lock(waiter.lock)

3      if waiter.task is mutex.owner or
         waiter.mutex is mutex then
         deadlock := true
       else

4        set_remove(task.waiters, waiter)

5        new_node(waiter, task, mutex)

6        set_add(find_proxy, waiter)

7        waiter.proxy_candidate := mutex.owner

       end_if

8      unlock(waiter.lock)
     end_for

9   return deadlock

   end_func
```

Program 3.4 presents the *prepare_waiters* function which determines, for a given location in a blocking chain, the set of waiters for which a new proxy must be found. This function assists Program 3.3 in updating SWR as specified by Equation 3.1 on page 35.

Line 1 initializes a *deadlock* variable to track if deadlock was detected for any waiters at this link of the m-blocking chain. The steps at line 2 loop over the set of waiters at this

location in the chain and acquire the spin lock that controls each waiter so that the waiter can be modified.

The steps at 3 perform deadlock detection by checking if the entire chain has been searched on behalf of a waiter. If a deadlock is detected then the waiter is not included in the *find_proxy* set and remains at its current location in the blocking chain in a searching state. At this point, there is obviously not a proxy for any of the waiters in the chain but it is necessary for the search to continue for some waiters in order to update SWR to satisfy Equation 3.9.

Line 4 removes the waiter from the task at the current location because the waiter is about to be moved to the next location in the chain.

Step 5 records the indirect waiting relation between *waiter* and *mutex* by creating a new node. The arguments to *new_node* are the waiter representing the task involved in the relation, the *via_task*, and the mutex involved in the relation. Here, the *via_task* argument to *new_node* is *task* because the chain is being traversed through *task*.

At line 6 the waiter is added to the *find_proxy* set to identify to the calling context, Program 3.3, that a new proxy needs to be found for this waiter. Line 7 identifies that the owner of the mutex is the current candidate for the proxy for this waiter. The *waiter.lock* is released at line 8 to allow the next waiter to be considered. Finally, line 10 notifies the calling context if deadlock was detected. The calling context may decide to ignore this value and allow deadlock to occur or it may abort the mutex lock operation.

The *sched_move_prepare*, *sched_move*, and *sched_destroy* functions, in Program 3.5, are responsible for informing the scheduling layer about waiter state changes. The calling context for these functions must hold the *pi_lock* of the *proxy_candidate* for the waiter because this lock protects the waiter's state and serializes callbacks involving the waiter.

Sched_move_prepare records a search state for a waiter and informs the scheduling layer. If a proxy has not already been found for a waiter then *Sched_move* records that a proxy has been found for a waiter and informs the scheduling layer. The state check in *sched_move* is

Program 3.5 Scheduling Layer Notifications

```
sched_move_prepare(waiter, old_proxy)

  waiter.state := PROXY_SEARCH
  sl.move_prepare(waiter, old_proxy)

end_func

sched_move(waiter, new_proxy)

  if waiter.state = WAITER_FINAL then
    return
  end_if

  waiter.state := PROXY_FOUND
  sl.move(waiter, new_proxy)

end_func

sched_destroy(waiter)

  if waiter.state is PROXY_FOUND then
    proxy := waiter.proxy_candidate
  else
    proxy := NULL
  end if

  sl.destroy(waiter, proxy)

end_func
```

necessary for line 5 of Program 3.3 to operate correctly.

Sched_destroy informs the scheduling layer that a waiter is no longer in use because its task has acquired a mutex or aborted an attempt to acquire one. The scheduling layer only cares about the actual proxy for a waiter and not its *proxy_candidate*. Therefore, we do not tell the scheduling layer about the *proxy_candidate* if a proxy was not found for the waiter.

The *new_node* function, in Program 3.6, is responsible for recording a waiting relation for a task. The steps at 1 retrieve a free node structure from a pool of free nodes. PMGT uses a pool of nodes instead of allocating a node using *kmalloc* because it is not possible

Program 3.6 Node Management

```
   new_node(waiter, via_task, via_mutex)

1    lock(node_pool_lock)
     node := dequeue(node_pool)
     unlock(node_pool_lock)

2    node.via_task := via_task
     node.via_mutex := via_mutex
     node.waiter := waiter

3    push(waiter.nodes, node)

4    set_add(via_mutex.nodes, node)

   end_func

   free_node(node)

     lock(node_pool_lock)
     enqueue(node_pool, node)
     unlock(node_pool_lock)

   end_func
```

to block in the context of a mutex operation because non-preemptable spin-locks are held. The *node_pool_lock* protects the pool of nodes. We currently have the kernel configured to panic if the node pool is exhausted and cannot allocate a new node. However, as discussed in Chapter 4 which considers the memory use of PMGT, showing that even under heavy load, the system uses at most a few thousand bytes worth of node structures. Further, the management of the pool could be extended to increase the pool size if a system exhibits a pattern of behavior using a significant portion of the available nodes.

The steps at 2 initialize the fields of the node. The *via_mutex* is the mutex involved in the waiting relation. The *via_task* is the waiter directly awaiting *via_mutex* which was traversed to reach *via_mutex*.

The waiting relations for each waiter are recorded in *waiter.nodes*. The relations are stored in a stack where the top of the stack is the waiting relation that involves the last

mutex encountered during searching the blocking chain. Nodes are only created as a proxy is searched for by a waiter and, thus, we know that the newly created node is the node closest to the proxy when this function is called. Therefore, line 3 pushes the newly created node on to the top of the stack.

Line 4 adds the node to the set of nodes associated with the mutex. This set allows PMGT to know, for a given mutex, all of the waiting relations involving that mutex.

The *free_node* function, in Program 3.6, puts a node in the pool of free nodes.

This completes the discussion of what happens and what algorithms are used when a task blocks on a mutex. Next we will consider what happens when a mutex is released by a task and a new owner must be selected from its waiters.

Wakeup_next_waiter

The *wakeup_next_waiter* function, in Program 3.7, is called by the PREEMPT-RT mutex implementation when a mutex is being released and a pending owner for the mutex must be selected and awakened. The pending owner is selected to receive the mutex but it does not actually take ownership until it starts to run and actually acquires the mutex. In the scenario depicted by Figure 3.8, task $T2$ is releasing mutex M and $T1$ is selected to be the pending owner for M. *Wakeup_next_waiter* is always executed by the owner of the mutex that is performing the mutex unlock operation. In the figure, $T2$ executes *wakeup_next_waiter* as *current* to release M.

Inside the Linux kernel code the *current* pointer refers to the task structure of the thread currently executing on the current CPU. In this scenario, the currently running process is the one releasing the mutex. Line 1 thus acquires *current.pi_lock* because, since *current* is releasing the mutex, *current* is no longer the proxy for any waiter and the waiters for which *current* is recorded as the proxy must be disassociated from *current*. Waiters currently associated with *current* must be moved to the pending owner. In the figure, $T1$ becomes the proxy for the tasks in S and A.

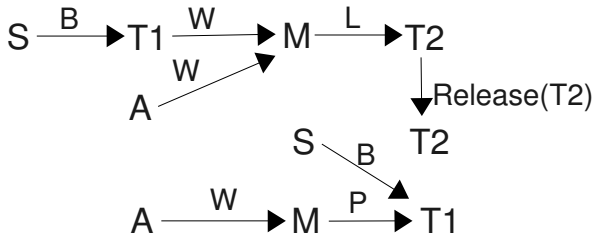


Figure 3.8. A Task Unlocking a Mutex

The steps at 2 find the best waiter on the mutex and remove that waiter from the mutex data structure. Recall that there is a *Best* function which is associated with every mutex to determine who the best task among a set of tasks is. The *mutex.top* callback of the mutex policy is responsible for applying the *Best* function to the set of waiters on the mutex and returning the best waiter. Additionally, the mutex policy determines how the data structures for the mutex are updated in the *mutex.remove* callback when the best waiter is removed. In the figure, $T1$ is the best waiter.

The best waiter on the mutex becomes the pending owner. A pending owner is the next task that will receive the mutex if a better task does not try to acquire the mutex before the pending owner does. If a better task does try to acquire the mutex before the pending owner then the mutex has been "stolen". The steps at 3 identify the pending owner and wake up the pending owner to cause the pending owner to try to take ownership of the mutex.

The steps at 4 destroy the waiter structure for the pending owner because the pending owner is no longer waiting on the mutex. First, the *waiter.lock* is acquired to allow manipulation of the waiter data structure. Second, the waiter must be removed from the set of tasks waiting on the proxy thread. We know that *current*, the owner of the mutex, is the proxy because *current* is performing a mutex operation and is obviously not blocked on a mutex at the same time. Thus, *waiter* is removed from the *current.waiters* set. Third, the scheduling layer is given a chance to perform cleanup of any data structures stored in the waiter structure by calling *sched_destroy*. Fourth, the waiter structure representing the

Program 3.7 Wakeup_next_waiter

```
    wakeup_next_waiter(mutex)

1   lock(current.pi_lock)

2   waiter := mutex.top()
    mutex.remove(waiter)

3   pendowner := waiter.task
    wakeup(pendowner)
    mutex.owner := pendowner
    mutex.state := PENDING

4   lock(waiter.lock)
    set_remove(current.waiters, waiter)
    sched_destroy(waiter)
    top_node := pop(waiter.nodes)
    set_remove(mutex.nodes, top_node)
    free_node(node)
    unlock(waiter.lock)

5   for each node in mutex.nodes do
      waiter := node.waiter
      set_remove(current.waiters, waiter)
      sched_move_prepare(waiter, current)
    end_for

6   sl.task_finalize(current)

7   unlock(current.pi_lock)
    lock(pendowner.pi_lock)

8   pendowner.waiter := NULL

9   for each node in
        mutex.nodes do
      waiter := node.waiter
      lock(waiter.lock)

10    if node.via_task is
          pendowner then
      pop(waiter.nodes)
      set_remove(mutex.nodes,
        node)
      free_node(node)
    end_if

11    waiter.proxy_candidate
        := pendowner
      set_add(pendowner.waiters,
        waiter)
      sched_move(waiter,
        pendowner)
      unlock(waiter.lock)
    end_for

12  sl.task_finalize(pendowner)
    unlock(pendowner.pi_lock)

  end_func
```

pending owner has exactly one node to represent its direct waiting relation with *mutex*. This node is removed from *waiter.nodes* and *mutex.nodes* and then disposed of by calling *free_node*. These steps contribute to the update of SWR and SPR in Equations 3.5 and 3.6. In the figure, the waiting relation $(T1, M)$ is removed.

The steps at 5 disassociate the set of waiters m-blocked on *current* from *current* because *current* is no longer part of the same chain as the waiters. These steps contribute to the update of SPR as described by Equation 3.6. They are examining the S_{task} and A_{task} portions of the blocking chain in Figure 3.8.

Line 6 tells the scheduling layer that we have finished modifying the set of waiters asso-

ciated with *current*. The steps at 7 drop the *pi_lock* for *current* and acquire the *pi_lock* for the pending owner because the waiters previously m-blocked on *current* are now m-blocked on the pending owner and these waiters need to be associated with the pending owner.

Line 8 clears the *pendowner.waiter* field because the pending owner is no longer waiting. This field could not be cleared earlier because it is protected by the *pendowner.pi_lock*. The steps at 9 iterate over all of the waiters that are being associated with the pending owner and acquire the lock for that waiter in order to modify it.

The waiters that are now m-blocked on the pending owner can be separated into two classes: waiters that were m-blocked on the pending owner before the mutex unlock operation, depicted by S_{task}, and those that were not, depicted by A_{task}. The set of waiting relations for each waiter which was m-blocked on the pending owner must be updated. PMGT identifies S_{task} by checking the *node.via_task* field to see which nodes reached the mutex by traversing the chain through *pendowner*. In Figure 3.8, M no longer lies between the tasks in S_{task} and their proxy, which is now $T1$. Therefore, the node at the top of the node stack for each element of S_{task} represents an invalid waiting relation and these invalid relations are removed by the steps at 10. A_{task} does not require any extra consideration because these tasks are still awaiting M since M is still between the tasks and their proxy. These steps satisfy Equation 3.5 which specifies the update to SWR.

The steps at 11 indicate that the pending owner is now the proxy for each of the waiters formerly m-blocked on *current*. These steps satisfy Equation 3.6 which specifies the update to SPR. The steps at 12 inform the scheduling layer that we modified the set of waiters associated with the pending owner and release the *pi_lock* of the pending owner to complete the transference of the mutex and cleanup.

Try_to_steal

Program 3.8 presents the *try_to_steal* function which is invoked when a task tries to steal a mutex. This function is called by the PREEMPT-RT mutex implementation during a

mutex lock operation before a task m-blocks on a mutex because the mutex has an owner. *Try_to_steal* returns true if the mutex was successfully stolen and the calling context assigns ownership of the mutex to the stealing task. Figure 3.9 depicts a task $T1$ stealing a mutex M whose pending owner is task $T2$. There is a dotted "W arrow" between $T1$ and M because $T1$ may be awaiting M when it tries to steal if it is unexpectedly awakened.

Program 3.8 Try_to_steal

```
    try_to_steal(stealer, mutex)

1     if mutex.state is not PENDING then
        return false
      end_if
      if mutex.owner is stealer then
        return true
      end_if

2     pendowner := mutex.owner
      lock(pendowner.pi_lock)

3     if not mutex.can_steal(mutex, stealer,
                             pendowner) then
        return false
      end_if

4     for each node in mutex.nodes do
        waiter := node.waiter
        set_remove(pendowner.waiters, waiter)
        sched_move_prepare(waiter, pendowner)
      end_for

5     sl.task_finalize(pendowner)

6     unlock(pendowner.pi_lock)
      lock(stealer.pi_lock)

7     for each node in mutex.nodes do
        waiter := node.waiter
        lock(waiter.lock)

8       if node.via_task is
            task then
          pop(waiter.nodes)
          set_remove(mutex.nodes, node)
          free_node(node)
        end_if

9       waiter.proxy_candidate
          := stealer
        set_add(pendowner.waiters, waiter)
        sched_move(waiter, stealer)

        unlock(stealer.lock)
      end_for

10    sl.task_finalize(task)
      unlock(task.pi_lock)

11    return true

    end_func
```

The steps at 1 check if stealing is actually possible. Stealing can only occur if the mutex has a pending owner and if the stealer is not already the pending owner. Stealing is always attempted by a waiter that is awakened and, therefore, when a waiter becomes the pending owner and is awakened it will try to steal the mutex. In this case, *try_to_steal* returns *true* immediately because the pending owner does not need to perform stealing. In Figure 3.9, $T2$ is a pending owner, indicated by the "P arrow", and the mutex is potentially steal-able

by all tasks other than $T2$.

The steps at 2 identify the pending owner and acquire the *pi_lock* for the pending owner to allow access to the PMGT data structures associated with it.

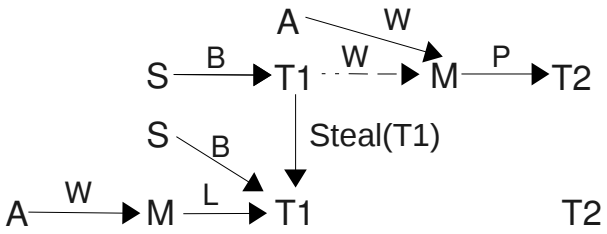


Figure 3.9. A Task Stealing a Mutex

Next, the algorithm must decide at 3 if *stealer* is allowed to steal the mutex. A mutex can only be stolen if *stealer* is better than the pending owner. However, PMGT does not know how to make this decision, and, instead, it asks the mutex policy to make the decision using the *mutex.can_steal* callback. The *mutex.can_steal* callback is responsible for checking if $stealer = Best(stealer, pendowner)$. In the scenario in the figure, it is assumed that $T1$ is better than $T2$ and, thus, that $T1$ is allowed to steal M.

If *stealer* is allowed to steal the mutex then it is necessary to make the stealer the new proxy for the chain. The steps at 4 disassociate all of the waiters on the mutex from the pending owner in preparation for assigning a new proxy to the waiters. At this point, the pending owner is no longer part of the m-blocking chain until it tries to acquire the mutex and m-blocks again. In the figure, the proxy for the tasks in S and the tasks in A must be changed.

At line 5, the scheduling layer is notified that PMGT has finished changing the set of waiters associated with the pending owner. The steps at 6 exchange the *pi_lock* of the pending owner for the *pi_lock* of the stealer to allow the waiters from the pending owner to be associated with the stealer.

The steps in 7-10 set the proxy for the waiters to be the stealer and inform the scheduling

layer of these changes. Additionally, any invalid waiting relations that result from the mutex being stolen are removed. These steps are equivalent to steps 9-12 of *wakeup_next_waiter*. See Program 3.7 for further details. In the figure, $T1$ is recorded as the proxy for the tasks in S and the tasks in A.

Line 11 returns true because the mutex was successfully stolen.

Remove_waiter

The *remove_waiter* function, in Program 3.9, is called by the PREEMPT-RT mutex implementation to remove a waiter from a mutex. There are two conditions under which this can occur: (1) the task represented by the waiter has aborted an attempt to acquire the mutex or (2) the task represented by the waiter unexpectedly woke up and stole the mutex and, since the task now owns the mutex, it is no longer a waiter. The first scenario is pictured in Figure 3.10 and the second scenario is pictured in Figure 3.9, if we assume that the waiting relation between $T1$ and M exists. In this function, The task represented by the waiter is denoted as *current* because it is the task that executes this function. In both figures, $T1$ is the task for which the waiter must be removed.

The steps at 1 remove the waiter from the set of waiters on the mutex by telling the mutex policy to remove the waiter using the *mutex.remove* callback. The data structure used to store waiters on a mutex is opaque to PMGT.

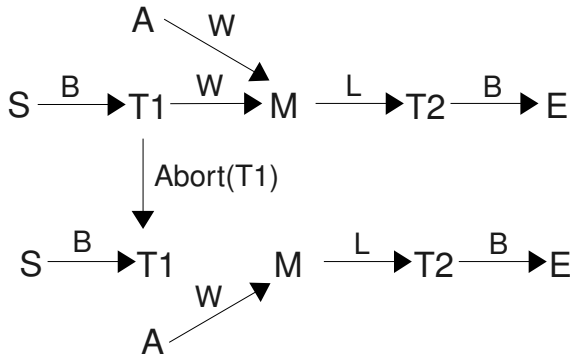


Figure 3.10. A Task Aborting an Attempt to Lock a Mutex

Program 3.9 Remove_waiter

```
   remove_waiter(mutex, waiter)

1    lock(current.pi_lock)
     mutex.remove(waiter)
     unlock(current.pi_lock)

2  retry:

3    lock(waiter.lock)
     proxy_candidate := waiter.proxy_candidate

4    if not trylock(proxy_candidate.pi_lock) then
       unlock(waiter.lock)
       goto retry
     end_if

5    node := peek(waiter.nodes)
     via_mutex := node.via_mutex
     if via_mutex is not mutex and
       not trylock(via_mutex.lock) then
       unlock(proxy_candidate.pi_lock)
       unlock(waiter.lock)
       goto retry
     end_if

6    set_remove(proxy_candidate.waiters, waiter)
     sched_destroy(waiter, proxy_candidate)

7    pop(waiter.nodes)
     while node is not NULL do
       lock(node.lock)
       node.waiter := NULL
       unlock(node.lock)
       node := pop(waiter.nodes)
     end_while

8    if via_mutex is not mutex then
       unlock(via_mutex.lock)
     end_if
     unlock(proxy_candidate.pi_lock)
     unlock(waiter.lock)

9    if mutex.owner is not current then
       move_waiters(mutex)
     end_if

     end_func
```

After the waiter is removed from the mutex it has to be unlinked from all PMGT data structures and destroyed. Unfortunately, in order to do that, spin-locks must be acquired out of order. A locking order is used throughout the OS to prevent the circular-wait condition for deadlock. To acquire spin-locks out of order, a locking transaction must be used instead to prevent the hold and wait condition. Therefore, the necessary spin-locks here are only acquired if they are all available. The *retry* label at line 2 indicates where the transaction begins.

First, the waiter needs to be removed from the set of waiters associated with its *proxy_candidate*. However, in order to find out who the *proxy_candidate* is, we have to lock the waiter structure itself in the steps at 3. The steps at 4 acquire the *proxy_candidate.pi_lock* because this lock allows PMGT to access the PMGT data structures for *proxy_candidate* to remove the waiter.

In Programs 3.1, 3.3, 3.7, and 3.8 waiters are transferred from one proxy candidate to a better candidate. The *waiter.lock* ensures that the *proxy_candidate* field of the waiter structure can be safely accessed. However, it does not insure that the waiter has been added to the data structures of the *proxy_candidate*. The steps at 5 ensure that the waiter is fully associated with its *proxy_candidate* by acquiring the controlling lock for the mutex m for which $Owns(proxy\_candidate, m)$ and $Awaits(current, m)$ hold. Mutex m is identified by the *via_mutex* field of the top node of the waiter's node stack.

In Figure 3.10, if E is empty then m will be M. However, the calling context of *remove_waiter* holds the lock controlling M and, thus, this lock is not acquired again. If E is not empty then m will be the mutex in E that is owned by *proxy_candidate* and the the lock controlling m will be acquired. In Figure 3.9, m is always M and no locking occurs because the lock for M is held by the calling context.

The steps at 6 remove the waiter ($T1$) from the set of waiters associated with the *proxy_candidate* and tell the scheduling layer that any data structures associated with that waiter should be cleaned up because it is going to be destroyed.

The nodes on the node stack of the waiter must be freed in order to completely destroy the record of $T1$ as a waiter and its relations. However, immediately freeing the nodes is difficult because each node is associated with a mutex and for each of these mutexes the lock controlling the mutex would have to be acquired in order to remove the node. Instead, the steps at 7 remove all of the nodes from the node stack and mark them as no longer being associated with a waiter. These "dead" nodes are still associated with their respective mutexes. The dead nodes for a mutex are cleaned up at the start of any mutex lock or mutex unlock operation on that mutex. Clean up consists of scanning the list of nodes on the mutex and returning any with a NULL *node.waiter* to the pool of free nodes. In Figure 3.10, the algorithm removes all of the nodes representing waiting relations of the form $(T1, m)$ where $m \in (\{M\} \cup E_{mutex})$. In Figure 3.9, the node representing $(T1, M)$ is removed.

The steps at 8 drop all of the spin locks because the waiter has been removed. If the waiter for *current* is being removed because *current* woke up unexpectedly and stole the mutex, as in Figure 3.9, then the function is finished because *try_to_steal* will have performed all other necessary updates. However, if the waiter for *current* is being removed because *current* is aborting a mutex lock operation, as in Figure 3.10, then *current* becomes the proxy for any waiters that are m-blocked on it. The waiters m-blocked on *current* are updated by *move_waiters* in the steps at 9. In Figure 3.10, $T1$ is recorded as the proxy for the tasks in S.

Move_waiters

When a task aborts a mutex lock operation it splits a blocking chain into two components: the blocking chain now associated with the task and the blocking chain associated with the mutex. The aborting task becomes the proxy for all of the tasks that are m-blocked on the aborting task and these tasks are no longer related to any of the tasks in the m-blocking chain associated with the mutex. The *move_waiters* function, in Program 3.10, updates the PMGT data structures to reflect these changes. The *mutex* argument to this function is

Program 3.10 Move_waiters

```
    move_waiters(mutex)

1     move_proxy := empty set

2     for each node in mutex.nodes do

3   retry:

4       lock(node.lock)
        if not node.waiter then
          unlock(node.lock)
          continue
        end_if
        if node.via_task is not current then
          unlock(node.lock)
          continue
        end_if

5       waiter := node.waiter
        if not trylock(waiter.lock) then
          unlock(node.lock)
          goto retry
        end_if
        proxy_candidate
          := waiter.proxy_candidate
        if not
            trylock(proxy_candidate.
            pi_lock) then
          unlock(waiter.lock)
          unlock(node.lock)
          goto retry
        end_if

6       top_node := peek(waiter.nodes)
        via_mutex := top_node.via_mutex
        if via_mutex is not mutex and
            not trylock(via_mutex.
            lock) then
          unlock(proxy_candidate.pi_lock)
          unlock(waiter.lock)
          unlock(node.lock)
          goto retry
        end_if

        /* All required mutexes are locked */

7       set_remove(proxy_candidate.waiters,
           waiter)
        waiter.proxy_candidate := current
        set_add(move_proxy, waiter)
8       if waiter.state is PROXY_FOUND then
          sched_move_prepare(waiter,
                             proxy_candidate)
          sl.task_finalize(proxy_candidate)
        end_if

9       do
          stale_node := pop(waiter.nodes)
          lock(stale_node.lock)
          stale_node.waiter := NULL
          unlock(stale_node.lock)
        while(stale_node is
          not node)

10      if via_mutex is not mutex then
          unlock(via_mutex.lock)
        end_if
        unlock(proxy_candidate.
          pi_lock)
        unlock(waiter.lock)
        unlock(node.lock)
      end_for

11    lock(current.pi_lock)
      for each waiter in move_proxy do
        set_add(current.waiters, waiter)
        sched_move(waiter, current)
      end_for
      sl.task_finalize(current)
      unlock(current.pi_lock)

    end_func
```

the mutex on which the mutex lock operation was performed. The task aborting the mutex lock operation executes this function and it is denoted as *current*. Figure 3.10, on page 70, depicts an abort scenario where the aborting task $T1$ becomes the proxy for the tasks in S.

Line 1 initializes a *move_proxy* set that stores the waiters for which *current* ($T1$) should become the proxy. At line 2, the algorithm iterates through all of the nodes that represent waiting relations that involve *mutex*. A subset of these nodes are associated with waiters that are m-blocked on *current* at the start of the abort. In the figure, the tasks in S are m-blocked on $T1$ at the start of the abort.

As in Program 3.9, spin locks are acquired out of order and as a result a locking transaction is used. The locks involved are: *node.lock*, *waiter.lock*, and *proxy_candidate.pi_lock*. Line 3 marks the start of the transaction, which is completed by line 7.

The steps at 4 examine the node to determine if it is associated with a waiter that is m-blocked on *current*. First, the *node.lock* is acquired to grant access to the node. Second, if the node is dead, as indicated by a null value in *node.waiter*, then it is ignored. Third, the node is skipped if *via_task* is not *current* because this indicates that the waiter associated with the node is not m-blocked on *current*. A waiter that is not m-blocked on *current* will still be part of the m-blocking chain associated with *mutex* and won't be affected by the abort. In Figure 3.10, nodes representing waiting relations of the form $(s, T1)$ where $s \in S$ will be processed. Those relations for tasks in A_{task} will not be via $T1$ and so be skipped.

The steps at 5 identify the waiter associated with the node. Additionally, they acquire the *waiter.lock* and the *proxy_candidate.pi_lock* in order to disassociate the waiter from its current proxy candidate. If either of these locks cannot be acquired then the locking transaction is restarted. The *via_mutex.lock* is acquired by the steps at 6 to ensure that the *proxy_candidate* field of the waiter structure is consistent with the set of waiters stored on the *proxy_candidate*.

The steps at 7 remove the waiter from its *proxy_candidate* and change the *proxy_candidate* to *current* because *current* is no longer blocked on a mutex. Additionally, the waiter is placed

in the *move_proxy* set for further updating.

If a proxy had been found for the waiter then the steps at 8 tell the scheduling layer that the proxy is changing. It is necessary to call the *sl.task_finalize* callback immediately because the waiters m-blocked on *current* could have different proxy candidates if multiple proxy searches are ongoing.

Every node representing a waiting relation involving *mutex* or a mutex that is between *proxy_candidate* and *mutex* is invalid because the m-blocking chain has been split. In the figure, the tasks in S are no longer related to M or any of the mutexes in E. Therefore, these nodes are removed from the node stack in the steps at 9. The nodes that are removed from the node stack are marked as dead because freeing the node requires a lot of additional concurrency control usage. These steps satisfy Equation 3.3 which specifies the update to SWR.

The steps at 10 release all of the spin locks acquired in the loop so that the next node can be processed and the steps at 11 make *current* the proxy for all of the waiters in the *move_proxy* set and inform the scheduling layer of these changes. These steps satisfy Equation 3.4 which specifies the update to SPR.

This brings the discussion of the PMGT algorithms supporting mutex operations, and thus Section 3.5.4, to an end. The algorithms presented fully implement the proxy accounting required to track waiter and proxy relations on the system, and to support the scheduling layer in the system which will use that information. The next section discusses how the PMGT layer can be used to implement the PI semantics in the PREEMPT-RT patch.

3.6 Priority Inheritance Implemented using PMGT

One way of demonstrating the correctness and feasibility of PMGT is to show that it can be used to implement the current system semantics. With that in mind, this section describes how PMGT can be added to the PREEMPT-RT patch and a configuration supporting the priority inheritance of its mutexes created.

The PI implementation in the PREEMPT-RT patch analyzes and tracks the set of waiters on a mutex in a way that is very similar to PMGT. A significant difference in the two approaches, however, is that the PI implementation does this analysis at m-blocking time as a way of determining what the priority of the proxy process ought to be but does not represent the proxy relations as such to the scheduling layer. The reason for this is that the priority scheduler only pays attention to priority values and so a proxy inheriting the best priority of itself and its waiters is sufficient.

Implementing PI semantics as a PMGT configuration involves using the proxy relations to have the proxy inherit the best priority of its waiters in the same way. Obviously, both the original PI and the PMGT-PI implementations must update the inherited priority when the set of waiters changes. When a proxy releases a mutex, its priority is adjusted back to its original value and the new proxy inherits the best priority of the waiters if that priority is better than its own.

The PMGT-PI implementation uses the priority mutex policy as the default mutex policy for the system in order to match the PREEMPT-RT PI mutex implementation which is priority based.

The PI and PMGT-PI implementations produce the same system behavior but differ slightly in the mechanisms they use. This difference in mechanisms means that the priorities of the waiting tasks are different under the two approaches because the PREEMPT-RT implementation updates the priorities of all tasks involved in a chain and the PMGT implementation updates the priority of only the proxy. The difference in the priorities of the waiters does not affect which tasks are run on the system because the waiters are not runnable in any case. However, it does affect the data observed by the instrumentation used by various tests of the PREEMPT-RT mutex implementation.

The correctness of the PMGT-PI implementation has been demonstrated by its correctly executing all RT mutex tests that come with the PREEMPT-RT patch. In most cases, the results of the tests are identical under both implementations. In a small number of cases,

we slightly modified the tests because the priority of the waiting tasks was different under the PMGT-PI implementation than expected but as already discussed this was not an error because the PMGT-PI implementation produces exactly the same set of scheduling decisions. This demonstrates that PMGT can be used under a scheduling layer other than HGS. It also demonstrates that the PMGT implementation is correct for this purpose.

In theory, one could consider changing the Linux schedulers, real-time and CFS, to directly use the proxy relations represented by the PMGT layer. This would be interesting on grounds of creating symmetry between the HGS and the PREEMPT-RT use of PMGT. It would also shift some of the use of the proxy information from blocking time to scheduling time which has implications for overhead. This overhead is necessary in the HGS context because HGS assumes arbitrary scheduling semantics. It is not necessary in the context of Linux's priority scheduling because update of priorities can be done at blocking time. Therefore, this thesis does not further consider these issues.

3.7 HGS Extensions

The fundamentals of HGS were well established in the IAS [20] [1]. However, there were a number of issues that had to be dealt with to enable HGS to be used for anything but the simplest of applications. First, the IAS mechanism for integration of HGS with Linux scheduling caused system instability. While specific reasons for the instability were not identified, the restructuring and expansion of the code described here has produced a system that is stable and robust when passing the existing 400 functional tests. Second, HGS did not correctly understand proxy relations involving threads executing on more than one CPU. HGS must operate correctly on SMP systems because they are now very common. Third, HGS failed to provide the PI concurrency control semantics assumed under PREEMPT-RT for all threads controlled by Linux schedulers. Solutions to these issues are part of the contributions of this thesis and are presented in this section. Finally, a proxy policy is discussed that incorporates all of the solutions to these issues when handling changes to the

system's set of proxy relations.

3.7.1 HGS Linux Integration

HGS provides a scheduling layer, with configurable schedulers, that aims to replace the priority based Linux scheduling layer. However, completely replacing the existing scheduling layer would be a very large change and the Linux community strongly prefers modifications presented in a more gradual manner. Further, re-implementing all of the priority scheduling semantics in Linux under HGS would be difficult. Therefore, we have taken an approach that re-uses significant aspects of the existing Linux scheduling layer by essentially wrapping HGS around the Linux model.

The Linux scheduling layer uses a stack model where different scheduling classes are consulted in a specific order for scheduling decisions. The stack consists of the real-time class on top, the CFS class in the middle, and the idle class on the bottom. Each scheduling class contains one or more schedulers to which tasks are assigned.

We adopt the scheduling semantics that HGS is always consulted before Linux when a scheduling decision is being made because many of the example applications that we have used with HGS require preferential scheduling. Additionally, HGS is consulted first because Linux always picks a task to schedule but HGS may not. Therefore, a lack of decision from HGS indicates that the Linux scheduling stack should be invoked. Finally, HGS is placed outside of the Linux scheduling stack to allow HGS to choose tasks that are also controlled by any Linux scheduler. This behavior is sometimes useful to give a system task, such as an interrupt handler, a reasonable priority based default behavior while also allowing the task to be scheduled on demand by HGS.

Unfortunately, the Linux scheduling layer assumes that all tasks on the system are assigned to a Linux scheduler. Therefore, it is not possible to indicate to the Linux scheduling layer, without modification, that a task should only be controlled by HGS. This constraint is problematic because, for some tasks, the HGS scheduler controlling the task assumes that

the task will not be scheduled by another scheduler.

Therefore, we now have an Exclusive Control (EC) Linux scheduling class that represents tasks under the exclusive control of HGS. The EC scheduling class provides HGS tasks with a Linux scheduler assignment that does not interfere with the schedulers in the HGS hierarchy. Usually, the EC scheduling class does not select tasks to be scheduled because HGS will schedule EC tasks as specified by the HGS hierarchy.

While tasks assigned to the EC class are handled by HGS outside the scope of the Linux scheduling stack most of the time, there are brief periods when this is not true. Tasks being added to or removed from the HGS hierarchy often have some administrative actions that must be completed even though the hierarchy data structures do not show them as the member of a group. In the case of a task being added to the hierarchy, often there is a small section of code that must be completed after the task is in the EC scheduling class but before the hierarchy can choose the task. Similarly, when a task is exiting, the task is removed from the hierarchy but it must continue to run for a few instructions in order to completely exit. The EC scheduling class is placed at the top of the Linux Scheduling Stack to give HGS tasks preference during these periods.

Since HGS tasks are generally preferred over Linux tasks, the EC scheduling class is placed at the top of the Linux Scheduling Stack to give EC tasks in an administrative period a chance to run before other Linux tasks. The EC scheduling class uses FIFO to choose tasks in administrative periods because administrative periods are generally short and, thus, a simple scheduler is sufficient.

Figure 3.11 depicts the organization of the HGS scheduling layer with the introduction of the EC scheduling class. At scheduling time, the HGS hierarchy is first evaluated and, if no decision is made, then the LSS is evaluated starting with the EC class.

Using the EC scheduler minimizes the number of changes required in the Linux scheduler layer in order for HGS to work cooperatively with the Linux scheduling layer. Therefore, the HGS patch size is smaller than it would be under other approaches, thus making it more

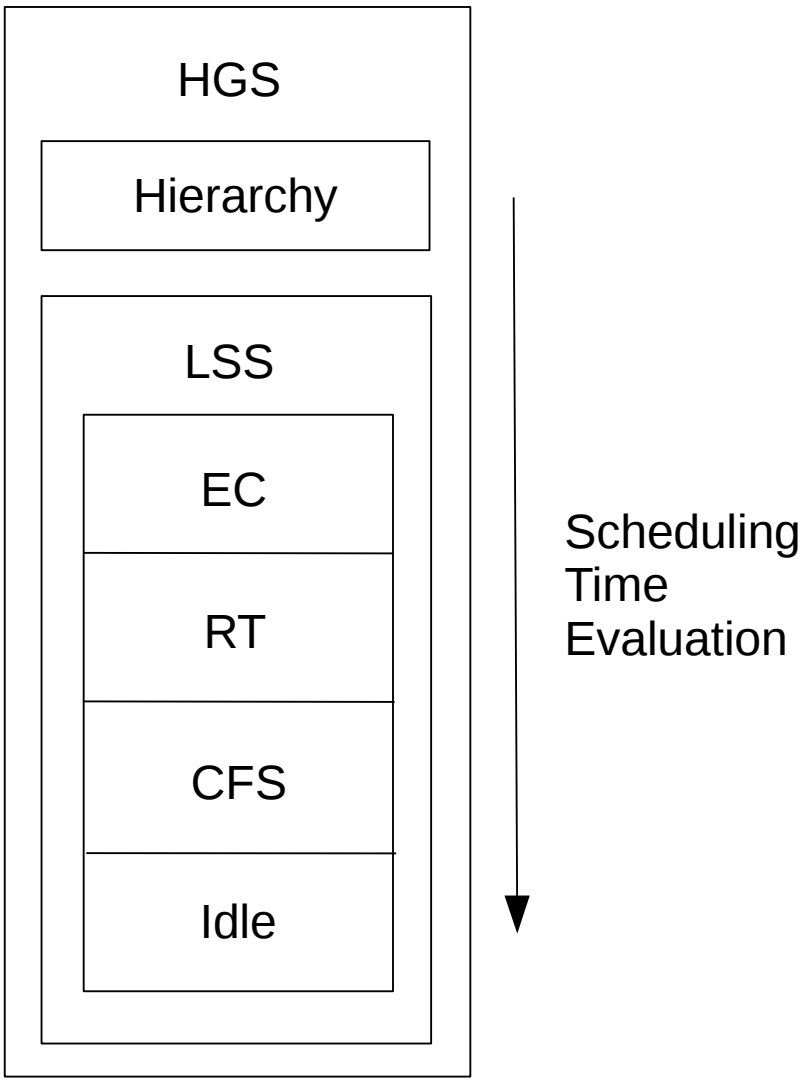


Figure 3.11. Integration of the LSS with HGS

acceptable to the Linux community.

3.7.2 SMP Proxy Selection

HGS uses an algorithm called Proxy Selection to handle proxy relations involving tasks with a wide range of scheduling semantics without having to consider the scheduling semantics involved. The version of Proxy Selection implemented in the IAS, was sufficient for

uniprocessor systems but it made it difficult for HGS schedulers to use data structures segregated on a per-CPU basis. This section discusses the extensions made to Proxy Selection to encourage the use of per-CPU data structures by HGS schedulers in an effort to increase scheduler parallelism.

Under HGS, multiple schedulers with widely different scheduling semantics can exist on a system at the same time. Consequently, at any given time, a proxy could have waiters controlled by schedulers with several different semantics. As the scheduling layer for the system, it is the responsibility of HGS to decide how a proxy's waiters effect the scheduling criteria of the proxy, even if the waiters have different semantics. The scheduling semantics of the proxy are partially derived from the scheduling semantics of its waiters. However, since the set of scheduling semantics that apply to the set of waiters can be arbitrary, the decision cannot be as simple as that used by PI since the waiters could be controlled by several different kinds of semantics. Instead, the Proxy Selection algorithm uses knowledge of the proxy relationships at scheduling time in a way that works for all situations.

Proxy Selection primarily consists of two components: the recording of proxy information about a task at locking time and the automatic substitution of the proxy for a task when the task is selected at scheduling time.

HGS implements a proxy policy, using the callbacks provided by PMGT, to update the proxy information about a task at locking time when proxy relations are created and destroyed. When a new proxy relation is created, the HGS Proxy Policy records the proxy in the HGS data structures that represent the task that is m-blocked on the proxy. Similarly, the proxy information for the task that was formerly m-blocked on the proxy is cleared when a proxy relation is destroyed.

There is one representation of a task for each group in which the task is a member. We call each representation of a task's membership in a group an *avatar* of the task. In addition to recording proxy information, avatars also record the scheduling criteria for a task as specified by the HGS scheduler associated with the group.

HGS schedulers select avatars at scheduling time from the group that they are associated with. An important part of how HGS can handle arbitrary sets of scheduling semantics is that a scheduler associated with a group can select an avatar which is m-blocked but which has a runnable proxy. When such an avatar is selected by a scheduler, HGS runs the proxy instead of the task the avatar represents. Thus, the m-blocking time for a task that the hierarchy wants to run is reduced because its proxy makes progress and thus removes the condition blocking the thread as quickly as possible.

It is not possible for a Linux scheduler to select a task that is m-blocked. Instead, PI has the proxy inherit the best priority of the waiters. However, this is possible only because all tasks use a common priority scheduling semantics, and one whose semantics permit inheriting the scheduling parameter of the "best" of the waiters. Other scheduling semantics, such as CPU share, explicit plan, or those using application state do not which is why the more general proxy accounting is required.

HGS does not allow an avatar to be selected on an arbitrary CPU. Under Linux, at a given instant, all tasks are assigned to a specific CPU and they can only run on that CPU. CPU assignment of a task can be changed, but the task cannot run on the new CPU until the change of assignment is complete. HGS also assigns each avatar to a specific CPU and each avatar can only be selected on that CPU. CPU assignment is beneficial because it allows per-CPU data structures to be utilized by schedulers to increase parallelism of scheduler invocations across CPUs.

Proxy Selection requires the CPU assignment of avatars to be modified based on the proxy for the avatar. If a task is not m-blocked then its avatars are assigned to the same CPU as the task because the avatar does not have a proxy. On the other hand, if a task m-blocks then the CPU assignment of its avatars may need to change. The task's avatars must be assigned to the CPU of the task's proxy in order for the proxy to be scheduled when any of the avatars are selected at scheduling time.

HGS tells the schedulers controlling the task's avatars to change the CPU assignment of

each avatar at locking time in order to assign the avatars to the CPU of the task's proxy and, thus, the schedulers are able to update their per-CPU data structures to reflect the change in CPU assignment. Similarly, the CPU assignment of a task's avatars can change when the proxy for a task changes or when a task no longer has a proxy because it has received ownership of a mutex on which it was previously m-blocked. The HGS Proxy Policy is responsible for coordinating avatar CPU assignment changes as a result of mutex operations and it is discussed further in Section 3.7.4.

An avatar is selectable by the CPU of the proxy immediately after its CPU assignment is changed. However, the CPU of the proxy is not obligated to select the avatar because the avatar maintains the same scheduling criteria that it had on the CPU of the waiter it represents. It is important that the avatar retain its scheduling criteria because, even though the avatar might be the best choice on the CPU of the waiter, the avatar may not be the best choice on the CPU of the proxy. Therefore, the proxy recorded by the avatar should not necessarily be immediately scheduled.

Avatar CPU assignments may also need to be changed due to: (1) the CPU assignment of a task being changed, (2) the CPU assignment of a proxy being changed, (3) a task's PMGT administrative period ending.

If the CPU assignment of a task is changed and the task is not m-blocked then the avatars for the task need to be assigned to the new CPU. If the task is m-blocked when its CPU assignment is changed then it's avatars do not change CPU assignment because they are assigned to the CPU of the task's proxy in order for HGS to be able to run the proxy in place of the task. Similarly, if a task is m-blocked and the CPU assignment of the task's proxy is changed then the task's avatars need to be assigned to the proxy's new CPU to guarantee that the proxy can still be run on behalf of the avatars. Finally, there are administrative periods in the PMGT code in which a task must run to update PMGT data structures when it has a proxy. At the end of the administrative period, the CPU assignments of the avatars for a task may need to be modified to reflect the changes made

to the PMGT data structures.

It is useful to keep some extra state information about each avatar to account for which CPU the avatar is assigned to. An active state and a blocked state are recorded for each avatar. The active state is the state that an avatar is currently in. This state determines if an avatar is selectable. The blocked state determines which state will become the active state when the avatar blocks.

The active state has four possible values: *RUNNABLE*, *BLOCKED*, *PROXY_SEARCH*, or *PROXY_FOUND*. The *RUNNABLE* state indicates that the avatar can be selected and the task that the avatar represents will be run. The *BLOCKED* state indicates that the avatar is blocked but it is not awaiting a mutex. In this state, the avatar cannot be selected. The *PROXY_SEARCH* state indicates that the avatar is awaiting a mutex but the proxy for the avatar has not yet been identified. An avatar in the *PROXY_SEARCH* state is not selectable. The *PROXY_FOUND* state indicates that a proxy has been identified for the avatar and the avatar can be selected. If the avatar is selected then the proxy for the avatar will be run.

Program 3.11 Hgs_cpu

```
func hgs_cpu(avatar)

  if avatar.state is RUNNABLE or avatar.state is BLOCKED then
    return avatar.task_cpu
  else
    return avatar.proxy_cpu

end_func
```

The blocked state has three possible values which have the same meaning as the corresponding active state values: *BLOCKED*, *PROXY_SEARCH*, or *PROXY_FOUND*. However, while these values have the same meaning as the corresponding active values, they do not influence selectability.

The *hgs_cpu* function, in Program 3.11, determines which CPU an avatar is assigned

to. The *avatar.state* field indicates the active state of the avatar. If an avatar is in the *RUNNABLE* or *BLOCKED* state then the avatar is assigned to the CPU of the task it represents. Otherwise, the avatar is m-blocked because it is in the *PROXY_SEARCH* or *PROXY_FOUND* state. If an avatar is m-blocked then it is assigned to the CPU of its last known proxy. In the *PROXY_FOUND* state this CPU will be the CPU of the proxy for the avatar since the proxy is currently known. The *avatar.proxy_cpu* defaults to the *task_cpu* so when a task first blocks, but before its proxy is found, the avatar will remain on the CPU of the task it represents until the CPU of the proxy is known.

Using the CPU of the last known proxy for the *PROXY_SEARCH* state is an optimization. If the proxy for a task is changing, and the tasks avatar's are thus temporarily entering the *PROXY_SEARCH* state, then the avatars for the task remain on the CPU of the old proxy until the new proxy has been found instead of needlessly moving back to the CPU of the task. The avatar is not selectable in the *PROXY_SEARCH* state and, thus, its CPU assignment isn't really of interest to a scheduler anyway.

Program 3.12 presents the pseudo-code for the *hgs_enqueue_task* function which is called by Linux when a task is becoming runnable. A task becomes runnable when it stops blocking on a resource, such as a mutex. Additionally, a task is temporarily not runnable while its CPU assignment or scheduling class assignment is modified and then becomes runnable again when the modification is finished. This function is responsible for ensuring that the avatars representing a task or related to a task are assigned to the appropriate CPU.

Linux uses a data structure called a "runqueue" to indicate which tasks are runnable. The name of the *hgs_enqueue_task* function refers to enqueuing because this function is called when the task is being placed on the runqueue to indicate that it is runnable. There is one runqueue for each CPU and each runqueue is protected by a non-preemptable spin-lock. The calling context of *hgs_enqueue_task* holds the runqueue lock for the CPU that the task is assigned to. The purpose of *hgs_enqueue_task* is to ensure that the avatars representing the task being enqueued and any avatars for which the enqueued task is the proxy are assigned

Program 3.12 Hgs_enqueue_task

```
   func hgs_enqueue_task(cpu, task, wake)

1    for each avatar in task.avatars do
       lock(avatar.ingroup.lock)

2      old_cpu := hgs_cpu(avatar)
       avatar.state := RUNNABLE
       avatar.task_cpu := cpu
       if old_cpu is not avatar.task_cpu then
         avatar.ingroup.scheduler.move(avatar, old_cpu, avatar.task_cpu)
       end_if
       unlock(avatar.ingroup.lock)
     end_for

3    for each avatar in task.blocked_avatars do
       lock(avatar.ingroup.lock)
       if avatar.proxy_cpu is not cpu then
         avatar.ingroup.scheduler.move(avatar, avatar.proxy_cpu, cpu)
         avatar.proxy_cpu := cpu
       end_if
       unlock(avatar.ingroup.lock)
     end_for

   end_func
```

to the correct CPU.

The *hgs_enqueue_task* function is provided with three arguments: the CPU on which the task is assigned, the task itself, and a Boolean variable *wake* that indicates if the task is waking up because it is no longer blocked. The *wake* argument is not currently used but it remains for the purpose of future changes and debugging purposes.

The steps at 1 look at each avatar that represents the task by iterating over *task.avatars*, the list of avatars for the task. Every group in HGS has a non-preemptable spin-lock associated with it and these spin-locks also control access to the avatars controlled by the group. The group controlling an avatar is identified by the *avatar.ingroup* field. Thus, the spin-lock for each group controlling the avatar is acquired in order to access the avatar's data.

The steps at 2 record the current CPU of the avatar's task, change the avatar's variables

to reflect that the task it represents is now runnable on *cpu*, and move the avatar to the task's CPU if the avatar is located on a different CPU. First, the CPU of the avatar is recorded by calling *hgs_cpu*. Next, the avatar's state is changed to *RUNNABLE* to reflect that the task it represents can now be scheduled. Additionally, the *avatar.task_cpu* is set to *cpu* to indicate which CPU the task it represents is assigned to. In order to cause the task to be scheduled, the avatar must be on the same CPU as the task. If it is not already then it is assigned to the CPU of the task. The *avatar.ingroup.scheduler* field points to the scheduler associated with the group controlling the avatar and the *avatar.ingroup.scheduler.move* callback tells the scheduler to change the CPU assignment of the avatar. Finally, the *ingroup.lock* is released in preparation for examining the next avatar.

If the CPU assignment of *task* changes and *task* is a proxy then the steps at 3 change the CPU assignment of the avatars for which it is a proxy. The *task.blocked_avatars* set contains all of the avatars that represent a task for which *task* is the proxy and the avatars in this set are moved to the CPU of the proxy if they are not already assigned to that CPU.

The *hgs_dequeue_task* function, in Program 3.13, is called when a task is no longer runnable and Linux is taking the task off of the runqueue for the CPU to which it is assigned. The calling context for *hgs_dequeue_task* holds the runqueue lock for the CPU. *Hgs_dequeue_task* accepts three arguments: the CPU assignment of the task, the task itself, and a Boolean variable *sleep* that indicates if the task is no longer schedulable because it is blocking on a resource. The primary goal of *hgs_dequeue_task* is to transition a task's avatars from the *RUNNABLE* state to one of *BLOCKED*, *PROXY_SEARCH*, or *PROXY_FOUND* when the task is blocking.

The *hgs_dequeue_task* function does nothing if a task is not blocking on a resource because, in this case, a task is only temporarily not runnable while its CPU assignment or scheduling class assignment is changed. The task will shortly become runnable again. The steps at 1 return immediately if the task is not blocking. If the task is blocking, the algorithm examines each avatar for the task and acquires the *avatar.ingroup.lock* that allows the

Program 3.13 Hgs_dequeue_task

```
   func hgs_dequeue_task(cpu, task, sleep)

1    if not sleep then
       return
     end_if

2    for each avatar of task do
       lock(avatar.ingroup.lock)

3      avatar.state := avatar.blocked_state

4      new_cpu := hgs_cpu(avatar)
       if cpu is not new_cpu then
         avatar.ingroup.scheduler.move(avatar, cpu, new_cpu)
       end_if

5      unlock(avatar.ingroup.lock)
     end_for

   end_func
```

avatar's data to be accessed in the steps at 2.

The line at 3 transitions the avatar into its blocked state. The blocked state is stored by the HGS Proxy Policy at locking time. The blocked state is stored because deducing the blocked state requires accessing PMGT data structures and that would further constrain concurrency.

Sometimes, when the avatar transitions to its blocked state then the CPU assignment of the avatar will also need to be changed. The steps at 4 tell the scheduler of the group controlling the avatar to change the CPU assignment of the avatar, if necessary. The CPU assignment of the avatar will only change if the avatar has a proxy and the avatar needs to move to the CPU of the proxy since the avatar's task is no longer schedulable. Primarily, this scenario occurs at the end of a PMGT administrative code execution period. The steps at 5 unlock the *avatar.ingroup.lock* because the loop is examining the next avatar.

The *hgs_enqueue_task* and *hgs_dequeue_task* functions cover all of the scenarios where the CPU assignment of an avatar must change outside of the HGS Proxy Policy callbacks

that occur at locking time.

It should now be clear that while the basic approach to PMGT was already established in the IAS for single processor systems, the work in this section was necessary to restructure and generalize it for use in SMP systems. Although it is now possible to segregate HGS scheduler data structures on a per-CPU basis, some constraint on concurrent execution of schedulers on each CPU remains and the system must be extended further to achieve full parallelism.

3.7.3 PI Compatibility

HGS must provide flexible concurrency control semantics to support the wide variety of scheduling semantics that can be implemented by HGS schedulers. However, at the same time, as the scheduling layer controlling the system, HGS is also responsible for the concurrency control semantics of the tasks controlled by the priority-based Linux schedulers because the Linux scheduling layer is re-used by HGS.

Section 3.7.2 discusses how HGS uses Proxy Selection to reduce the m-blocking time of waiters by scheduling the proxy for the waiter instead of the waiter. Proxy Selection could, in theory, produce behavior identical to PI for the Linux schedulers. However, adapting these schedulers to use Proxy Selection would require extensive modification because Linux schedulers do not allow threads that are blocked on a mutex to be selected.

Therefore, HGS uses a PI approximation at locking time to reproduces PI behavior without any modifications to the internal Linux scheduling API. This is only an approximation because the presence of non-priority based tasks that are exclusively controlled by HGS forces some compromises to be made. HGS implements a proxy policy that incorporates the PI approximation. Additionally, HGS provides a default system mutex policy that produces standard priority semantics for selecting the next owner of a mutex when the waiters for a mutex are all controlled by a priority-based Linux scheduler. If EC tasks are involved then the HGS Mutex Policy gives preference to the EC tasks.

At locking time, when a new proxy relation is created for a waiter and a proxy, HGS considers the Linux scheduling class that the waiter is assigned to. If the waiter is assigned to a Linux scheduling class other than the EC scheduling class then the waiter is added to a priority sorted list stored on the proxy. The proxy inherits the best priority of the waiters on that list. Later, when the waiter is no longer waiting, the waiter is removed from the list and PI is again performed for the proxy.

The Linux scheduling class that the proxy is assigned to must also be considered because if the proxy is under the control of the EC scheduling class then inheriting a priority from a task controlled by Linux's priority schedulers will have no effect because the EC scheduling class and the HGS hierarchy do not look at priorities. In this case, a proxy assigned to the EC scheduling class is temporarily re-assigned to either the real-time or CFS Linux scheduling classes, depending on the inherited priority. Thus, the proxy becomes selectable by the priority schedulers using the inherited priority and the m-blocking time of priority-based Linux waiters may be reduced. During this period, the proxy is still selectable by HGS.

Another scenario of interest occurs when an EC task has a non-EC proxy. In this case, the non-EC proxy cannot inherit a priority from the EC task because the EC task doesn't have a priority to inherit. However, since we allow the HGS hierarchy to select a task from any scheduling class, the non-EC task will be scheduled by the HGS hierarchy when the EC task is selected by an HGS scheduler and, thus, the m-blocking time of the EC task may be reduced.

The *hgs_pmgt_adjust* function is invoked by HGS at locking time, during several HGS Proxy Policy callbacks, to adjust the scheduling criteria of a task because the set of tasks for which the task is a proxy has changed. The pseudo-code for this function is presented in Program 3.14. The calling context for this function must hold the runqueue lock for the CPU that the task is assigned to because this lock controls the Linux scheduling data modified by this function.

The steps at 1 perform PI for the task. The HGS Proxy Policy maintains a sorted list of

Program 3.14 Hgs_pmgt_adjust

```
   func hgs_pmgt_adjust(task)

1    if task.sched_policy is EC then
       if proxy.ppd.pi_waiters is not empty then
         task.prio := priority of the first waiter on
           proxy.ppd.pi_waiters
       end_if
     else
       task.prio := min(
         priority of the first waiter on proxy.ppd.pi_waiters,
         task.normal_prio)
     end_if

2    if task.ppd.pi_waiters is empty and
       task.sched_policy is EC then
       task.sched_class := EC
     else
       if task.prio is a real-time priority then
         task.sched_class := RT
       else
         task.sched_class := CFS
       end_if
     end_if

end_func
```

priority-based Linux tasks that are m-blocked on a proxy. The waiter with the best priority is the first element on the list and the priority of this waiter is the priority that the task may need to inherit. This list is easily accessible at locking time because it is stored in the PMGT data structures associated with the proxy. Recall that PMGT allows the system's proxy policy to store arbitrary proxy policy data in the PMGT data structures associated with the proxy for this purpose. The *proxy.ppd* field is the location of the proxy policy data and the *proxy.ppd.pi_waiters* field is the list of priority-based Linux tasks for which *task* is the proxy.

A task's *sched_policy* field indicates the Linux scheduling class that the task should be controlled by when it is not m-blocked. If a task's *sched_policy* is EC then the task inherits

the best priority of its waiters. Note that an EC task does not use priorities and, thus, it does not have a priority to consider in the PI calculation. If a task is not controlled by the EC scheduling class then it is assigned the minimum value of its normal priority, which is not influenced by inheritance, and the best priority of its waiters. The minimum value is used because lower priorities are better under Linux.

The steps at 2 consider the scheduling class of the task. If it is an EC task and it does not have any priority-based waiters then it remains controlled by the EC scheduling class. Otherwise, if the task is not an EC task, then the task is placed in the scheduling class to which its inherited priority belongs. Thus, an EC task will be placed under the control of a priority-based scheduling class if it inherits a priority and the priority-based scheduling class can schedule the task and indirectly reduce the m-blocking time of tasks with better priorities.

The PI approximation used by the HGS Proxy Policy provides handling of proxy relations involving EC tasks and non-EC tasks. However, it is also vital to consider how EC tasks and non-EC tasks interact when they are awaiting the same mutex. HGS specifies a default system mutex policy called the HGS Mutex Policy which adheres to the priority semantics of the non-EC tasks while also considering the often more important EC tasks.

The default HGS Mutex Policy orders the tasks awaiting a mutex it controls in order to optimize the selection of the next owner. Waiters are stored in a priority sorted list in which waiters with the same priority are stored in FIFO order. EC waiters are associated with a single priority that is better than all of the priorities within the allowed Linux priority range. The special priority assigned to EC tasks is stored in the PMGT data structure for the waiter and so it does not effect scheduling because it is only visible to the HGS Mutex Policy. EC tasks appear first in the list in FIFO order because they have better priorities than all non-EC tasks and they are all assigned the same priority. Non-EC tasks are inserted in the list using their actual priority. When a mutex is unlocked the HGS Mutex Policy simply picks the first waiter on the list to receive the mutex.

The PI approximation used by HGS helps to maintain behavior similar to standard Linux scheduling semantics under PREEMPT-RT while also allowing HGS to cooperate with the Linux Scheduling Stack.

The primary responsibility of a mutex policy is selecting the next owner of the mutex, from its current set of waiters, when the current owner unlocks the mutex. When all waiters on the mutex are controlled by priority semantics, this it is easy to see that the best waiter is the one with the best priority. However, under HGS the set of semantics under which the set of waiters is controlled can be an arbitrary mix of all semantics used to control tasks on the system since any tasks can use mutexes associated with Linux services. This is why the default mutex waiter selection policy under HGS is two-tiered. First, it gives preference to tasks controlled by HGS, which are those in the EC class. Among those tasks, it implements a FIFO policy since choosing any other is impossible without some idea of the HGS SDF semantics being used in the HGS hierarchy. If no EC tasks, those controlled by HGS, are waiting for the mutex, then selection is by Linux priority.

It is important to note that system implementers are in complete control of the scheduling semantics that can be configured into the HGS hierarchy. If the developers know enough, perhaps by limiting the set of schedulers used, then it is perfectly possible to implement a mutex waiter selection policy that is far more subtle. For example, if deadline scheduling is used in the HGS hierarchy, then it would be simple to create a policy giving preference to those with deadlines, those under HGS control, and to select by deadline among the HGS waiters. It is important to note that PMGT and HGS are frameworks permitting the implementation and configuration of a wide range of scheduling and concurrency control semantics, but it is the responsibility of the system architects to ensure that the semantics they select are feasible.

3.7.4 The HGS Proxy Policy

SMP Proxy Selection and PI compatibility are dependent on taking actions in response to changes to the system's set of proxy relations. HGS implements a proxy policy, called the HGS Proxy Policy, that incorporates these features. This section discusses the algorithms used by the HGS Proxy Policy: (1) hgs_pmgt_task_init, (2) hgs_pmgt_waiter_init, (3) hgs_pmgt_move_prepare, (4) hgs_pmgt_move, (5) hgs_pmgt_task_finalize, and (5) hgs_pmgt_destroy.

Recall that PMGT provides two initialization callbacks that a proxy policy can implement: *task_init* and *waiter_init*. The *task_init* callback allows a proxy policy to initialize data structures that are used to store information about a task when it is a proxy. This callback is invoked when a task is created. The *waiter_init* callback allows a proxy policy to initialize data maintained about a task when it is m-blocking. This data is stored in the waiter data structure that PMGT uses to represent m-blocked tasks in order to make it easily accessible during PMGT callbacks.

Hgs_pmgt_task_init

The HGS Proxy Policy uses the *hgs_pmgt_task_init* function in Program 3.15 for its *task_init* callback. This function initializes a priority sorted list of waiters, *task.ppd.pi_waiters*, that is used for PI compatibility to determine the scheduling class and priority of the task when it is a proxy. The algorithm used to change the scheduling class and priority of the proxy is discussed in Program 3.14. *Hgs_pmgt_task_init* also initializes a set of avatars that tracks which avatars are m-blocked on the task because the CPU assignment of these avatars may need to change in response to a change in the CPU assignment of the proxy. More details about this change in CPU assignment are provided in Program 3.12.

Program 3.15 Hgs_pmgt_task_init

```
func hgs_pmgt_task_init(task)
  task.ppd.pi_waiters := empty set
  task.blocked_avatars := empty set
end_func
```

PMGT allows a proxy policy to store proxy policy data in PMGT data structures because these data structures can be easily accessed during PMGT callbacks. The algorithms presented here use *ppd* to indicate that proxy policy data is being accessed. HGS stores the priority-based waiters list, identified as *task.ppd.pi_waiters*, in the proxy policy data because the list is only accessed during PMGT callbacks. The set of avatars for which the task is a proxy, identified as *task.blocked_avatars*, is stored directly in the task structure because it is accessed outside of PMGT callbacks and PMGT data structures are difficult to access from outside of the concurrency control context provided by the PMGT callbacks. As a result, some concurrency control for scheduling data structures must be utilized within PMGT callbacks to update this set but this is generally easier than accessing PMGT data structures from outside of the context provided by the PMGT callbacks.

Hgs_pmgt_waiter_init

The HGS Proxy Policy uses the *hgs_pmgt_waiter_init* function, in Program 3.16, for its *waiter_init* callback. This function copies data into the proxy policy data for the waiter that is needed for the PI compatibility and SMP Proxy Selection features during PMGT callbacks.

Program 3.16 Hgs_pmgt_waiter_init

```
   func hgs_pmgt_waiter_init(waiter, task)
1    waiter.ppd.prio := task.prio
     waiter.ppd.sched_policy := task.sched_policy

2    task_rq_lock(task)
     for each avatar in task.avatars do
       set_add(waiter.ppd.avatars, avatar)
       avatar.state := PROXY_SEARCH
       avatar.proxy_cpu := avatar.task_cpu
     end_of
     task_rq_unlock(task))

   end_func
```

The steps at 1 store the priority, *task.prio*, of the task and the Linux scheduler that

controls the task when it is not a proxy, *task.sched_policy*, on the waiter because these variables determine if the waiter should be listed on the *pi_waiters* list of the proxy and where in the list it should be located.

The steps at 2 create a list of the avatars for the m-blocking task in the proxy policy data for the waiter, *waiter.ppd.avatars*, to make the avatars easily accessible during PMGT callbacks. Additionally, the steps at 2 mark each of the task's avatars as searching for a proxy by setting *avatar.state* to *PROXY_SEARCH* and indicate that the avatar is performing the search from its current location on the CPU of the m-blocking task by setting *avatar.proxy_cpu*. Note that the list of avatars for a task is protected by the runqueue lock of the runqueue for the CPU that the task is assigned to. Therefore, the runqueue for the task is locked during these steps using the *task_rq_lock* function provided by Linux.

Hgs_pmgt_move_prepare

After a waiter is initialized because the task it represents is m-blocking, PMGT will search for a proxy for the waiter. The proxy policy views the search through invocations of the *move_prepare* and *move* callbacks. The *move_prepare* callback indicates that a waiter is no longer blocked on its current proxy and that a new proxy must be found. The *move* callback indicates that a proxy has been found for a waiter.

Program 3.17 presents the pseudo-code of the *hgs_pmgt_move_prepare* function used by the HGS Proxy Policy for the *move_prepare* callback. This function updates the avatars for the task represented by the waiter to reflect that they no longer have a proxy and that a new proxy is being searched for. The waiter and the previous proxy for the waiter, *old_proxy*, are provided as arguments.

The steps at 1 remove the waiter from the *old_proxy.pi_waiters* list if it represents a non-EC task because the task is no longer m-blocked on *old_proxy* and no longer contributes to the inherited priority of *old_proxy*. Line 2 acquires the runqueue for *old_proxy* to allow the avatars associated with the waiter to be safely removed from the *old_proxy.blocked_avatars*

Program 3.17 Hgs_pmgt_move_prepare

```
   func hgs_pmgt_move_prepare(waiter, old_proxy)

1    if waiter.ppd.sched_policy is not EC then
       set_remove(old_proxy.ppd.pi_waiters, waiter)
     end_if

2    task_rq_lock(old_proxy)

3    for each avatar in waiter.ppd.avatars do
       lock(avatar.ingroup.lock)
       avatar.blocked_state := PROXY_SEARCH
       if avatar.state is not RUNNABLE then
         avatar.state := avatar.blocked_state
       end_if
       unlock(avatar.ingroup.lock)
4      set_remove(old_proxy.blocked_avatars, avatar)
     end_for

5    task_rq_unlock(old_proxy)

end_func
```

set.

The steps at 3 transition all of the avatars associated with the waiter to a searching state by iterating over *waiter.ppd.avatars*. The lock for the group that the avatar is in, *avatar.ingroup.lock*, is held while each avatar is modified because this lock controls access to the avatar. The blocked state for each avatar is recorded as *PROXY_SEARCH* because a new proxy is being searched for on behalf of the waiter. The actual active state of the avatar is only changed to match the blocked state if the task is not runnable because a task may be running as it searches for its own proxy. If the active state of the avatars for a searching task was changed to the *PROXY_SEARCH* state then the task would be stuck in the middle of a search because HGS would think that the task could not be scheduled. Line 4 disassociates each avatar from *old_proxy* by removing it from *old_proxy.blocked_avatars*.

Finally, line 5 cleans up by unlocking the runqueue lock that was earlier acquired.

Hgs_pmgt_move

The *hgs_pmgt_move* function in Program 3.18 is used by the HGS Proxy Policy for the *move* callback. It is provided two arguments: the waiter for whom a new proxy has been found and the new proxy that was found. This function is responsible for recording the proxy for each avatar and, if the task represented by the avatar is not runnable, assigning the waiter's avatars to the CPU of the proxy. This is a crucial step in SMP Proxy Selection that allows an avatar to be selected on the CPU of the proxy so that the proxy can be run in place of the avatar.

The steps at 1 add the waiter to the *new_proxy.pi_waiters* list if it represents a non-EC task because the task now contributes to the inherited priority of *new_proxy*. EC tasks are treated differently. Their avatars are later placed on the *new_proxy.blocked_avatars* list to identify to the proxy the set of avatars for which it is the proxy.

The runqueue lock for *new_proxy* is acquired by the steps at 2 to enable the set of blocked avatars, represented by *new_proxy.blocked_avatars* to later be modified. Additionally, this lock ensures that the CPU assignment of the proxy does not change while we are examining the avatars. The steps at 3 look at each of the avatars and acquire the group lock that controls each avatar in order to modify it.

The steps at 4 assign the *PROXY_FOUND* state to each avatar's blocked state because the proxy for these avatars has been found. The active state of the avatar is only changed to the *PROXY_FOUND* state if the avatar is not runnable because the task represented by the avatar may be executing administrative actions that must be completed. If the active state of the avatars is changed to *PROXY_FOUND* then, at scheduling time, the proxy for the avatar will be run in place of the avatar's task and the task will not make progress. The CPU of the proxy is also recorded because the avatar will be assigned to the CPU of the proxy if the task it represents is not runnable. Finally, the avatar is placed in the *new_proxy.blocked_avatars* set because *new_proxy* may need to know the set of avatars for which it is a proxy.

Program 3.18 Hgs_pmgt_move

```
   func hgs_pmgt_move(waiter, new_proxy)

1    if new_proxy.ppd.sched_policy is not EC then
       set_add(new_proxy.ppd.pi_waiters, waiter)
     end_if

2    task_rq_lock(new_proxy)

3    for each avatar in waiter.ppd.waiters do
       lock(avatar.ingroup.lock)

4      old_cpu := hgs_cpu(avatar)
       avatar.blocked_state := PROXY_FOUND
       if avatar.state is not RUNNABLE then
         avatar.state := avatar.blocked_state
       end_if
       avatar.proxy := new_proxy
       avatar.proxy_cpu := task_cpu(new_proxy)
       set_add(new_proxy.blocked_avatars, avatar)

5      if old_cpu is not hgs_cpu(avatar) then
         avatar.ingroup.scheduler.move(avatar,
           old_cpu, avatar.proxy_cpu)
       end_if

6      unlock(avatar.ingroup.lock)
     end_for

7    task_rq_unlock(new_proxy)

end_func
```

If the CPU assignment of the avatar must be changed to the CPU of the proxy then the steps at 5 tell the scheduler of the group that the avatar is in to change the CPU assignment of the avatar. The steps at 6 unlock the spin-lock that controls access to the avatar and move on to the next avatar. Line 7 cleans up by unlocking the runqueue lock that was earlier acquired.

Hgs_pmgt_task_finalize

PMGT may invoke the *move_prepare* callback several times consecutively to disassociate waiters from a task which was previously their proxy. Similarly, PMGT may invoke the *move* callback several times consecutively to associate multiple waiters with a proxy. PMGT provides a proxy policy callback called *task_finalize* that enables a proxy policy to change the scheduling criteria of a task for an entire set of *move_prepare* or *move* invocations instead of changing the criteria of the task for each invocation.

Program 3.19 Hgs_pmgt_task_finalize

```
func hgs_pmgt_task_finalize(task)
  task_rq_lock(task)
  hgs_pmgt_adjust(task)
  task_rq_unlock(task)
end_func
```

The HGS Proxy Policy uses the *hgs_pmgt_task_finalize* function in Program 3.19 for its *task_finalize* callback. This function is responsible for updating the inherited priority and scheduling class of a task in response to changes to the set of waiters for which the task is a proxy. It accomplishes this by calling the *hgs_pmgt_adjust* function discussed in Program 3.14. The runqueue lock of the CPU that the task is assigned to is held when *hgs_pmgt_adjust* is called because *hgs_pmgt_adjust* accesses the task's Linux scheduling data.

Hgs_pmgt_destroy

Eventually, every mutex lock operation performed by a task results in the task acquiring the mutex or aborting the operation. In both cases, the waiter data structure representing the task must be disposed of. PMGT provides a *waiter_destroy* callback that tells a proxy policy when a waiter data structure is being disposed of.

Program 3.20 presents the *hgs_pmgt_destroy* function which is used by the HGS Proxy Policy for the *waiter_destroy* callback. This algorithm must ensure that the waiter is no longer involved in PI and it must assign any avatars for the waiter to the CPU of the task.

Two arguments are provided to this callback: the waiter to remove and the proxy for the waiter. The proxy argument is NULL if the waiter is currently searching for a proxy.

Program 3.20 Hgs_pmgt_destroy

```
   function hgs_pmgt_destroy(waiter, proxy)

1  if proxy is not NULL then

2    if waiter.ppd.sched_policy is not EC then
       set_remove(proxy.ppd.pi_waiters, waiter)
     end_if

3    task_rq_lock(proxy)
     hgs_pmgt_adjust(proxy)
   end_if

4  for each avatar in waiter.ppd.avatars do
     if proxy is not NULL then
         set_remove(proxy.blocked_avatars, avatar)
     lock(avatar.ingroup.lock)
     avatar.blocked_state := BLOCKED
     if avatar.state is not RUNNABLE then
       avatar.state := avatar.blocked_state
     end_if
     if avatar.proxy_cpu is not avatar.task_cpu then
       avatar.ingroup.scheduler.move(avatar,
         avatar.proxy_cpu, avatar.task_cpu)
     end_if
     unlock(avatar.ingroup.lock)
   end_for

5  if proxy is not NULL then
     task_rq_unlock(proxy))
   end_if

end_for
```

At line 1 the algorithm checks if the proxy is NULL. If the waiter does not currently have a proxy then it cannot be involved in PI. Therefore, no PI calculations are performed. When the proxy is not NULL and the waiter represents a non-EC task then the waiter is removed from the list of waiters that contribute to the priority of the proxy in the steps at 2.

The steps at 3 acquire the runqueue lock for the proxy for two reasons: to potentially ad-

just the priority of the proxy in response to a task being removed from the *proxy.ppd.pi_waiters* list and to disassociate all of the avatars of the waiter from the proxy. The priority of the proxy is adjusted by calling *hgs_pmgt_adjust*. The avatars of the waiter are disassociated from the proxy later on by removing all of the avatars from the *proxy.blocked_avatars* set.

Next, *hgs_pmgt_destroy* examines each of the waiter's avatars in the steps at 4. First, each avatar is removed from the *proxy.blocked_avatars* set because they are no longer associated with the proxy. Second, the group lock that controls each avatar is acquired to allow the avatar to be moved to the CPU of the task associated with the waiter. At the start of *hgs_pmgt_destroy*, the avatar will have a blocked state of *PROXY_SEARCH* or *PROXY_FOUND* which indicates that it is m-blocked. The *BLOCKED* state is assigned to *avatar.blocked_state* to indicate that the avatar is no longer m-blocked. If the avatar is not runnable then *avatar.state* is also changed. Finally, the scheduler controlling the avatar is informed if the CPU assignment needs to change.

The steps at 5 clean up by releasing the runqueue lock for the proxy if it was earlier acquired.

The algorithms in this section demonstrate how to implement a proxy policy that combines PI compatibility with SMP Proxy Selection. Additionally, these algorithms demonstrate that it is significantly easier to implement a proxy policy than it is to implement a concurrency control layer with custom semantics. The PI approximation used by these algorithms to provide PI compatibility is only slightly more complicated than the steps performed by PI at locking time. However, SMP Proxy Selection adds significant complexity at locking time in exchange for the greater control provided by HGS at scheduling time.

3.8 Guided Execution

We propose that using scheduling to control application behavior by explicitly implementing the scheduling semantics in a customized scheduler is often clearer than trying to emulate those semantics using a combination of priorities and concurrency control. De-

terministic testing provides an example domain where directly implementing the desired semantics in a specialized scheduler is much clearer than using a combination of priorities and concurrency control.

Testing software that involves concurrency is notoriously difficult because concurrency creates a situation where many different interleavings may exist and thus different parts of the code may be executed in many different orders. One of the most common approaches to testing concurrent software is the "stress" or "soak" approach which executes as many instances of software use as possible assuming if enough instances are executed then all possible scenarios will have been covered at least once [10]. A slightly less crude approach is non-deterministic testing which adds instrumentation so that the set of scenarios covered can be known and compared to the set of desired scenarios [8] [19]. Deterministic testing refers to methods that exert control over what scenario executes [5].

Deterministic testing represents the testing problem as one of covering all possible sequences of synchronization operations, SYN-Sequences, that can be produced by a set of application threads executing concurrently. A synchronization operation is any operation that interacts with another thread. The crucial point in this approach is that the set of activities between synchronization operations is not constrained, only the way in which the synchronization operations of each thread are interleaved with those of other threads to produce the SYN-Sequence of the application.

The particular SYN-Sequence produced by the application depends on a wide range of system context including: timer interrupt arrivals, other interrupt arrivals, and the pattern of context switches. As a result, the probability of producing some possible SYN-Sequences is considerably larger than producing others. Deterministic Testing is particularly useful for producing rare SYN-Sequences. A limitation of deterministic testing is that all of the possible SYN-Sequences have to be specified for the testing to be complete and that can be difficult because there can be a great many of them. Also, it can be difficult to know all of the possible SYN-Sequences.

Deterministic testing is our preferred method of testing PMGT and HGS because there are many rare SYN-Sequences we would like to be able to test and these sequences are difficult to produce using non-deterministic testing. Many of these rare SYN-Sequences are unlikely to occur because they depend on: a particular sequence of events that occur randomly, the reception of a signal, or an interrupt at a specific point in a sequence of actions. For the PMGT and HGS tests, the mutex lock and unlock are the synchronization operations.

Often, deterministic testing is implemented by using concurrency control or priority manipulation to constrain thread execution to produce a specific sequence of synchronization operations. The Guided Execution environment instead uses a scheduler that can follow an explicitly specified schedule to produce any desired SYN-Sequence.

It is important to note that the Guided Execution approach can produce any SYN-Sequence but it does not address the implications of all possible thread interleavings within the synchronization operations themselves.

In the Linux kernel, three kinds of concurrency are present and must be controlled in various situations: interrupt, thread, and physical. Interrupt concurrency is not relevant to testing RT mutexes because their use in interrupt context is not permitted. Thread concurrency refers to arbitrary preemption of one thread by another sharing the same CPU. The PMGT implementation of RT mutexes disables preemption in all but the case where an m-blocking thread walks down the m-blocking chain to find its proxy. In that case, preemption is permitted between links in the chain. The Guided Execution methods used to control interleavings can in principle be applied to this portion of the RT mutex code in the kernel but that is not yet part of our testing suite.

Physical concurrency refers to threads executing the same section of code on different CPUs. RT mutexes are implemented using code sections that are only physically concurrent and these sections are frequently used. To test these sections, threads would have to execute the same section at the same time on different CPUs. This sort of testing requires fine grained control of precisely when two or more threads begin executing that is difficult to achieve

with scheduling because it is below the time scale of scheduling decisions. Additionally, Guided Execution would have no control over the interleaving of instructions executed by the threads during this period because it controls thread concurrency, not physical concurrency. Therefore, Guided Execution cannot support tests that depend on controlling physically concurrent execution.

A model checker, such as SPIN [11], might be used to perform analysis of the aspects of RT mutexes affected by physical concurrency but a model has not yet been developed for PMGT. A model for PMGT should perhaps be developed towards this purpose.

A GE prototype was discussed in [20]. This section presents a fully featured implementation of GE and illustrates its use by discussing the PMGT-HGS test suite which consists of over four hundred tests.

A GE test consists of two parts: a schedule and waypoints inserted into the application code. A GE schedule consists of a series of elements which pair a waypoint with the thread that has reached it. Usually, a waypoint is a label placed in an application telling the SDF a thread's location. In addition, GE provides a small set of waypoints that are automatically reached when the state of a thread changes, such as when the thread blocks on a mutex or receives a signal. These automatic waypoints are called conditions to distinguish them from normal waypoints. Thus, a GE schedule defines the order in which different threads in an application should reach different program states.

A GE schedule is similar to a SYN-Sequence in that it defines the order in which threads should execute. Unlike a SYN-Sequence, a GE schedule is expressed in terms of waypoints that are completely independent of any particular synchronization operation implementation. GE can thus in principle be used to implement a wide range of tests unrelated to SYN-Sequences. A GE schedule can trivially be mapped to a SYN-Sequence by associating each synchronization operation with a waypoint, which is what we have done for the tests in the PMGT-HGS test suite.

Threads in an application declare their names when they place themselves under the

control of GE and the schedules for the PMGT-HGS tests are expressed using those names. When a thread executes a waypoint, this is the announcement to GE that a waypoint has been reached. GE then decides, using the schedule, if the thread should continue to execute or if a different thread should start to execute. When a waypoint is reached and GE decides to stop executing the current thread, GE forces the system to make a scheduling decision and the system consults GE about which thread should run. GE then picks the next thread in the schedule to run.

Figure 3.12 shows an example of a GE schedule. This schedule creates the m-blocking chain depicted in Figure 3.13. The *schedule* list specifies the pairings of threads and waypoints that imposes an ordering on the execution of the program. The $t1$, $t2$, and $t3$ lists specify the operations performed by the threads in the schedule. Section 3.8.1 discusses tools used by the PMGT-HGS tests to automatically convert these specifications into a test executable program but GE can be used by applications that do not use these utilities by directly calling GE.

```
schedule = [
       t1(wp=lock1)
       t2(wp=lock2)
       t2(cond=block1)
       t3(cond=block2)
       ]

t1 = [
       lock(mutex=1)
       wp(name=lock1)
       ]

t2 = [
       lock(mutex=2)
       wp(name=lock2)
       lock(mutex=1)
       ]

t3 = [
       lock(mutex=2)
       ]
```

Figure 3.12. An Example PMGT Test Configuration

The first element of the schedule specifies that a thread $t1$ reaches a waypoint called *wp_lock*1. The $t1$ operations list starts with a "lock" operation that acquires mutex 1 and finishes with a "wp" operation that indicates to GE that the schedule should be advanced to

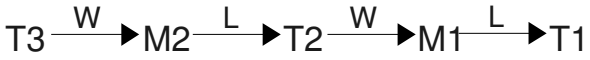


Figure 3.13. M-Blocking Chain Produced by GE

the next location. When the "wp" operation is executed, GE transitions to the next location in the schedule. Similarly, The second element of the schedule specifies that $t2$ reaches a waypoint called wp_lock2. The $t2$ operations list starts with corresponding "lock" and "wp" operations.

The next element of the schedule, $t2(cond = block1)$, has a "lock" operation in the operations list for $t2$ but it does not have a "wp" operation because this element of the schedule specifies a condition that is automatically detected by GE. GE automatically advances the schedule when $t2$ m-blocks on mutex 1. Finally, the last element of the schedule indicates that GE should detect when $t3$ m-blocks on mutex 2. GE allows the threads to run without restriction after the last element of the schedule has been fulfilled.

As a testing framework, GE is most useful if it can be applied to many different situations. The deterministic control performed by GE can be applied to kernel threads as well as user-space threads because system threads interact with the scheduler in the same way as user space threads. Sometimes, system threads are necessary to test functionality that is not exposed to user space. However, user-space tests are preferred, when possible, because in general they are easier to implement.

Additionally, GE can be used to test threads under the control of any Linux scheduler to provide a wider range of possible tests. Threads that are not controlled by the EC scheduler are not selected by GE in order to allow these threads to behave as they would under the standard Linux schedulers. Some special handling of non-EC threads is necessary because the standard Linux schedulers could run these threads when it is not their turn to run in the

schedule. Therefore, GE places all threads that are not supposed to run in a blocked state and makes them runnable again when it is their turn to run under the GE schedule.

3.8.1 Testing Proxy Management

PMGT is in charge of tracking the waiting relationships among a set of threads holding or awaiting a set of mutexes. Therefore, the PMGT tests are expressed at the level of RT mutex operation sequences. These are the SYN-Sequences of Deterministic Testing theory and are controlled by GE.

Each PMGT test describes a context in which it begins, a mutex operation, and the context in which it ends. All but the last operation in the SYN-Sequence describing a test is establishing the beginning context of the test while the last synchronization operation is the one being tested. The ending context is the state of the test after the last synchronization operation. The correctness of a given test is evaluated by examining the sequence of actions performed and the structure of the ending context.

The beginning context of all tests is essentially a m-blocking chain involving a set of threads and mutexes. This provides the context in which the mutex operation being tested must execute. The nature of proxy accounting dictates that different contexts for a given operation can result in quite different sets of actions. There are, therefore, a reasonably large number of scenarios required to test individual operations in all possible contexts.

The mutex operation evaluates the context in which it starts and chooses which actions it takes based on that analysis. The actions taken by a mutex operation can differ a lot depending on the context in which it occurs. In the simplest case, the beginning context of a thread attempting to lock a free mutex is simply the mutex. In contrast, a thread attempting to lock a mutex held by another thread which is itself in the middle of a complex and branching blocking chain can present a vastly different analysis and modification problem for PMGT.

User-space threads interact with PMGT through the PI mutex implementation in the

PThreads library, which, in turn, relies on the PI-futex implementation provided by the **futex** system call. The **futex** system call creates a PREEMPT-RT mutex to represent each user level PI-futex under contention. The set of mutex operations provided by PThreads is sufficient for most tests but a few scenarios do have to be implemented in a kernel module rather than user space code.

For example, PThreads does not expose all of the operations necessary to test the deadlock support in PMGT. PThreads does not support the return of a deadlock error code from **futex**. Thus, it is not possible to write user space tests that abort locking operations when deadlock is detected and these tests were instead done in a kernel module. Also, **futex** does not provide support for aborting attempts to lock a mutex due to a signal. When a signal occurs during **futex**, the system call is automatically restarted. Therefore, it was not possible to implement tests aborting lock operations due to receiving a signal under PThreads and these were instead done in a kernel module.

There are 31 different PMGT tests in the PMGT-HGS test suite. Only three of these tests are implemented as a kernel module. Therefore, many tests can be accomplished using user-space tests despite the limitations imposed by PThreads.

At a practical level, we wanted to create a testing framework which would support iterative expansion of the test suite, make it as easy as possible to understand each scenario and the test code implementing it, and make it as easy as possible to adapt to any future changes to the RT mutex code. In addition, the testing framework also had to enable us to create a set of tests corresponding to each scenario to cover all possible execution contexts.

The approach we chose uses a C template for all user tests and another for all kernel module tests. A configuration file specifies each scenario and a test generation tool is used to generate the C code implementing a specific test from the relevant template. The configuration file specification of the SYN-Sequence for the test consists of: (1) the GE schedule and (2) a sequence of operations and waypoints for each thread.

Several layers of Python scripts control the generation of code for the set of tests, their

compilation, their execution, and the evaluation of their output as success or failure.

This approach thus permits iterative expansion and refinement of the tests by modifying configuration files or generating new ones. The automation of test generation, execution, and evaluation makes it practical for use as a regression test suite. The completeness of the test suite is not formally proven in any way but the existing set of tools is well suited to implementing any set of scenarios. If formal methods were used to generate a complete set of scenarios [13], this framework could be used to implement them.

3.8.2 HGS Testing

The scenarios used to test PMGT can also be used to test HGS but there are added parameters that also must be considered for each scenario because HGS has to deal with a number of factors besides the blocking relations that exist between a set of threads and mutexes. Each added parameter adds a dimension to the parameter space that exists for PMGT and, thus, each PMGT scenario becomes several tests in the HGS testing space. In addition to using the existing PMGT scenarios, some new HGS specific scenarios are also necessary.

The added parameters are CPU assignment, scheduling class, priority, and the number of HGS memberships. Each of these values has a set of possible values. For example, while every thread has a specific CPU assignment, for testing purposes, we generally care whether two threads are on the same CPU or different CPUs but not which CPUs they are executing on. Similarly, while there are several scheduling classes in Linux, for testing purposes we only care whether a given thread is in the EC class supporting HGS or in one of the Linux classes. With respect to priority we don't care about the specific values. Instead the parameter represents how the relative values of the proxy and its waiters affect whether the proxy's priority must be adjusted through inheritance, as discussed in Section 3.7.3. In a given scenario, we want to make sure that our implementation works for threads that have no HGS memberships, are members of a single HGS group, or are members of multiple

groups. This is the issue that the HGS memberships parameter addresses.

The total number of tests required to cover the parameter space is a practical concern. A simplistic approach might define the set of tests as every possible combination of every parameter for every thread in a scenario. However, a more careful analysis shows that the size of the parameter space can be constrained in several ways.

One way that we can constrain the total number of tests is by observing that the set of interesting threads in most scenarios is a subset of all of the threads. A proxy is always of interest because it is the only thread that may be runnable and it is influenced by all of its waiters. Additionally, if the proxy changes then both the old proxy and the new proxy are of interest. One or more waiters will also be of interest. This set is determined by the combination of parameters for the waiters at the start of the scenario and if that combination changes during the scenario. By limiting the set of tests to only consider the combination of parameters for the set of interesting threads in such scenarios we considerably reduce the total number of tests.

Similarly, the total set of tests can be constrained by converting a literal enumeration for a given parameter value into a smaller number of qualitative values. For example, a literal interpretation of the CPU assignment parameter would explicitly represent the CPU on which each thread in a scenario was executing. Instead, we can represent this parameter with just two values representing whether the threads of interest and the proxy are executing on the same or different CPUs. CPU assignment of the interesting threads is important because some scenarios involve avatars moving from one CPU to another during Proxy Selection, as discussed in Section 3.7.2. This parameter does not have to consider explicit CPU assignment because movement of an avatar is unnecessary if the waiter(s) are on the same CPU as the proxy. Similarly, when the waiter(s) and the proxy are on different CPUs then avatar movement is involved but which CPUs are involved is not important.

Even after every effort to constrain the parameter space, there are still a large number of tests that must be defined and performed. The large number of tests motivated the

creation of automated tool support. This was done in two phases. First, the existing test generation tool was extended to handle the new parameters such as scheduling class and CPU assignment. Second, a scenario generation tool was created to process basic scenario configuration files containing set of values for each of the scenario parameters. The new tool generates a test configuration file from the basic scenario for each combination of parameter values specified.

Some new basic scenarios are also necessary because, in some contexts, the change of one thread's parameter values could effect another thread without any mutex operations taking place. For example, a call changing the priority of a thread, the scheduling class of a thread, or the CPU assignment of a thread can require action at the HGS representation level. Thus, calls changing parameter values are now synchronization operations. The tests for these new scenarios are still being developed.

There are 417 HGS tests and, like the PMGT tests, these tests are fully automated. The PMGT-HGS test suite provides significantly more thorough coverage than the test suite provided with PREEMPT-RT which contains only 28 tests.

Chapter 4

Evaluation

Both PMGT and PREEMPT-RT have greater execution overhead than the non-preemptable concurrency control present in standard Linux. The increase in execution overhead of PREEMPT-RT over standard Linux is generally perceived as reasonable in real-time systems where the benefits of more precise behavior control justify it. PMGT both adds additional execution overhead to PREEMPT-RT and shifts some of the overhead from locking time to scheduling time. The benefit of PMGT is that it generalizes the PREEMPT-RT semantics to support essentially arbitrary scheduling semantics rather than being limited to priority scheduling. The key point is how wide the range of applications will be for which the benefit of generalized semantics justifies the increased overhead. In this chapter experiments are presented that demonstrate the correctness of the PMGT and HGS implementations described in Chapter 3 and evaluate the performance implications of the methods used to implement the flexibly configurable PMGT and HGS layers.

PMGT adds representations of the relations between the elements in an m-blocking chain in order to generalize the integration of scheduling and mutex semantics beyond the Priority Inheritance of PREEMPT-RT. This also creates more memory overhead than PREEMPT-RT which only has to arrange for priority to be inherited at locking time. Often, PMGT requires the relation information to track the identity of the proxy for an m-blocked task

when the m-blocking chain that the task is part of is being modified. For example, when a task is m-blocking on a mutex or a mutex is being released.

It is important to make the memory and execution overhead of PMGT small enough to not significantly limit the range of applications. Memory overhead reduces the amount of memory available to applications and execution overhead decreases the amount of time that an application spends actually performing its intended activities.

There are two types of execution overhead of interest: mutex operation overhead and scheduling overhead. Mutex operation overhead is analyzed in Section 4.1 by measuring the time a task takes to m-block under various conditions and the time required to unlock a mutex. Scheduling overhead is examined in Section 4.2 by measuring the time required to make a scheduling decision for pipeline processing applications with varying numbers of tasks being controlled.

This chapter is concerned with the execution overhead of three approaches to the integration of scheduling and concurrency control: PREEMPT-RT, PMGT-PI, and HGS. PREEMPT-RT provides a reference point for comparison because it is widely accepted and it is the foundation on which PMGT-PI and HGS are built. PMGT-PI is of interest because it demonstrates how much overhead is added to the PREEMPT-RT mutex implementation by PMGT when the same PI semantics are implemented. HGS demonstrates the overhead of a scheduling layer that uses proxy relations instead of inheriting priority.

Memory overhead is evaluated in Section 4.3 by considering the number of data structures representing waiting relations maintained by PMGT under PMGT-PI when the system is under heavy load. Any scheduling layer could be used because an m-blocking chain has the same PMGT memory overhead under any scheduling layer.

All of the histograms presented in this section use logarithmic vertical scales to ensure that both buckets with a small number of samples and buckets with a large number of samples are visible. Some of the histograms are created during the post-processing of experimental data and others are created during the experiment as the data is gathered, in order

to decrease the instrumentation effect. Exact variance and median values are reported for histograms created in post-processing. Variance and median values are estimated for histograms gathered during the experiment. The variance and median estimates are calculated after the experiment is complete by using the bucket data. The median is calculated by multiplying the count for each bucket by the midpoint of the bucket and accumulating a list of these values for all buckets. The middle value of the list is the median estimate. Similarly, the variance is estimated by creating a set of estimated samples for each bucket using the midpoint of the bucket and the count for the bucket. The difference between each estimated sample and the mean of the data is used as normal in calculating variance. We could use a calculation method based on each sample as it is gathered but that would require squaring the value of each sample during the experiment. Since the whole point of directly gathering a histogram is to minimize the instrumentation effect, this was considered less desirable than the approximation method.

4.1 Mutex Operation Overhead

This section discusses experiments which measure mutex operation overhead by gathering data about the m-blocking time and the time required to release a mutex. Guided Execution is used to execute two scenarios involving various threads and mutexes that exercise the most commonly used portions of the mutex lock and mutex unlock operations. Each scenario is executed 5000 times to gather a distribution of values.

All of the tasks in these experiments are managed by Guided Execution but they are scheduled by CFS. This means that Guided Execution may block a task to ensure that a specific scenario unfolds as desired, but it does not function as a scheduler. This configuration is preferred because the scheduling semantics of the tasks are the same as they would be under PREEMPT-RT and PMGT-PI without Guided Execution being present.

4.1.1 M-Blocking Time

This experiment measures the m-blocking time for tasks which we define as the time between when the task tries to acquire a mutex owned by another task and the time that it blocks. It is important to note that the mutex lock code disables interrupts and preemption early in its execution and re-enables them shortly before the blocking thread blocks. Under PMGT, the blocking tasks looks for its proxy by walking down an m-blocking chain, during which the implementation provides two options: keep preemption and interrupts disabled or permit them at each step in the chain. In theory, permitting preemption and interrupts increases concurrency but delays the time to complete PMGT accounting by finding the proxy. In contrast, disabling preemption and interrupts decreases concurrency but minimizes the time to find the proxy. Either approach might be preferred under particular circumstances by a system architect which is why the choice of which to use is a kernel configuration parameter. However, as discussed later in this section, under the operating conditions we used for testing the behavior of the system is not significantly different under either option.

For this experiment, Guided Execution is used to execute the scenario in Figure 4.1. In this figure, task $T4$ is m-blocking on mutex $M3$ which is owned by task $T3$. Task $T4$ must traverse the m-blocking chain to determine that $T1$ is its proxy. This scenario is executed 5000 times for PREEMPT-RT, PMGT-PI, and HGS to measure the difference in overhead between these implementations. Each of these configurations was tested under three sets of conditions: (1) on an otherwise idle dual CPU system, (2) on a dual CPU system with competing load, and (3) on a single CPU system with competing load. A kernel compile using the "-j4" option to the "make" command was used for competing load.

The data for this experiment is gathered by emitting a Datastream event when $T4$ tries to acquire $M3$ (start) and a second Datastream event is emitted after $T4$ has found its proxy and shortly before it blocks (stop). Additionally, Datastream events are gathered that record the context switch and interrupt activity of $T4$ in order to determine if $T4$ is preempted or interrupted while it is searching for its proxy. In the figure, if preemption or interrupts are

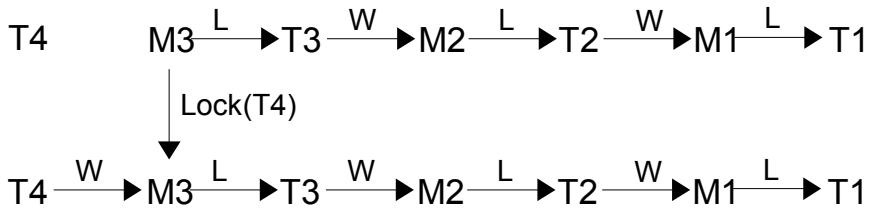


Figure 4.1. An M-Blocking Scenario

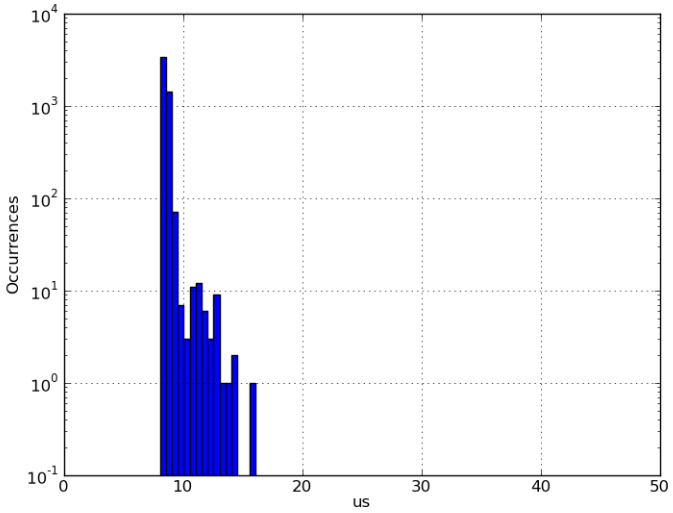
permitted, $T4$ experiences two intervals during which this could happen as it transitions from $M3$ to $M2$ and from $M2$ to $M1$ as it searches for its proxy $T1$. In post-processing the elapsed time between the start and stop events was determined by taking the difference between their time stamps and that datum was inserted into a histogram. Post-processing can check to see if preemption or interrupts occurred by looking for those events between start and stop.

4.1.1.1 Idle

Measuring the m-blocking time on an otherwise idle system provides a base line for how long a task takes to m-block because influences such as interrupts, preemption, and physical contention are minimized.

Figure 4.2 shows the distribution of m-blocking times for PREEMPT-RT on an otherwise idle system. This histogram contains the expected 5000 values. It has 100 buckets that have a size of 0.50. The minimum value seen is 8.03 and the maximum value is 16.04 microseconds. There are no underflow and overflow values. The average value is 8.47 and the median is 8.40. The standard deviation is 0.42, which indicates that the data are closely clustered around the average value.

Post-processing indicated that no interrupt or preemption activity occurred between any two start and stop events and so we did not create a corrected histogram filtering those



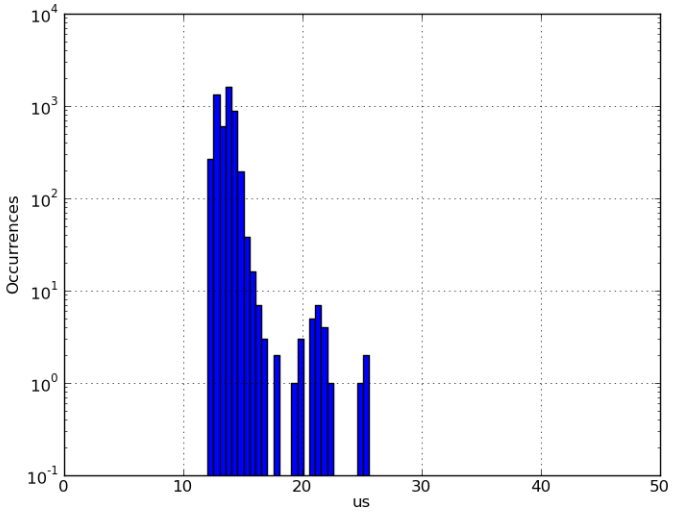
Count	5000
Buckets	100
Bucket Size	0.50
Min	8.03
Max	16.04
Underflow	0
Overflow	0
Average	8.47
σ	0.42
Median	8.40

Figure 4.2. PREEMPT-RT M-Blocking Idle

effects out. Since the system was otherwise idle it is not surprising that no interrupt or preemption activity occurred.

Figure 4.3 displays the distribution of m-blocking times under PMGT-PI. The minimum and maximum values for this histogram increased to 12.29 and 25.34 microseconds, respectively. The average increased to 13.56 from 8.47 for PREEMPT-RT, indicating an increase in overhead resulting from PMGT. The standard deviation increased to 0.90 from 0.42 for PREEMPT-RT indicating that the behavior for PMGT-PI is slightly more variable. Post-processing indicated that no interrupt or preemption activity contributed to this data.

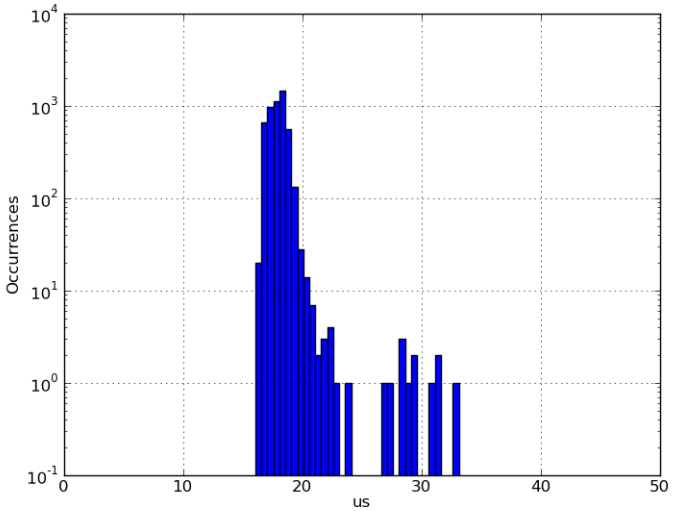
The results of this experiment when executed under HGS are illustrated by Figure 4.4. The HGS hierarchy for this experiment was the simple Guided Execution hierarchy which consists of a top level group with a Guided Execution group as a member. The results thus represent the overhead of using HGS with minimal scheduler addition. The minimum and maximum values increased to 16.33 and 32.84 microseconds, respectively. Additionally, an increased average of 17.93 compared to 13.56 for PMGT-PI is significant. The standard deviation, in contrast, was essentially unchanged at 0.91 compared to 0.90. There are two factors that contribute to the increased overhead. First, the increased complexity of the



Count	5000
Buckets	100
Bucket Size	0.50
Min	12.29
Max	25.34
Underflow	0
Overflow	0
Average	13.56
σ	0.90
Median	13.64

Figure 4.3. PMGT-PI M-Blocking Idle

HGS versions of *move_prepare* and *move* compared to that of the PMGT-PI. Second, the HGS versions require more complex concurrency control, the per-CPU runqueue lock and the HGS group locks, which increases the base level of overhead and increases the probability of physical contention. No interrupt or scheduling activity was present in this data.



Count	5000
Buckets	100
Bucket Size	0.50
Min	16.33
Max	32.84
Underflow	0
Overflow	0
Average	17.93
σ	0.91
Median	17.97

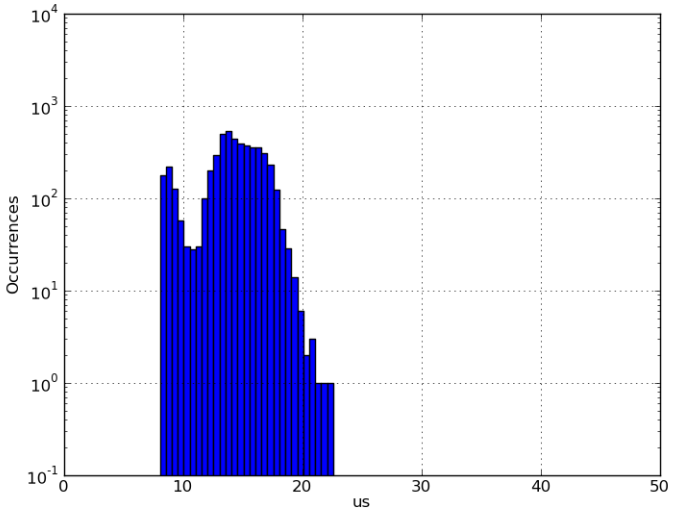
Figure 4.4. HGS M-Blocking Idle

4.1.1.2 SMP Load

Since a task can be interrupt-able at times when it is searching an m-blocking chain for its proxy, it is useful to know how much this is affected by interrupts when there are many interrupts occurring. When an interrupt occurs, the interval being measured is increased by the interrupt service time. In addition, some interrupts can result in context switches which further increase the measured intervals. Thus, experiments using competing load were added to the study of m-blocking overhead to see how often interrupts and preemption occur during the searching of the m-blocking chain. When they occur, the events produced during the experiment make it possible to detect them and correct for the majority of the time that they add to the measured intervals for finding the proxy.

Adding competing load produces significantly different results. Figure 4.5 depicts the results of repeatedly executing the m-blocking scenario depicted in Figure 4.1 on a dual CPU system while simultaneously compiling a kernel. The "-j4" option is specified to the kernel compile to increase the load of concurrently executing processes. Visually, when compared to 4.2, this histogram appears to have a greater variance and there are two peaks instead of one. Physical contention for the spinlocks used internally by the PREEMPT-RT mutex implementation could cause some variation in the appearance of the histogram but this does not seem to entirely account for the change when results from later experiments are considered.

Under load, the minimum remains roughly 8 microseconds but the maximum has increased from 16 to 160. However, this is not surprising when preemption and interrupts are considered. The data-set contains 9 instances of interrupt and preemption activity, five of which correspond to the five overflow values for this histogram. When the distribution is corrected for interrupts and preemption, the maximum is reduced to 24.90. The average remains essentially constant at about 14 microseconds but the standard deviation is significantly reduced from 4.04 to 2.52, which suggests that the intervals during which interrupts occurred were contributing significantly to the variation in behavior. However, since only



Count	5000
Buckets	100
Bucket Size	0.50
Min	8.11
Max	160.75
Underflow	0
Overflow	5
Average	14.16
σ	4.04
Median	14.26

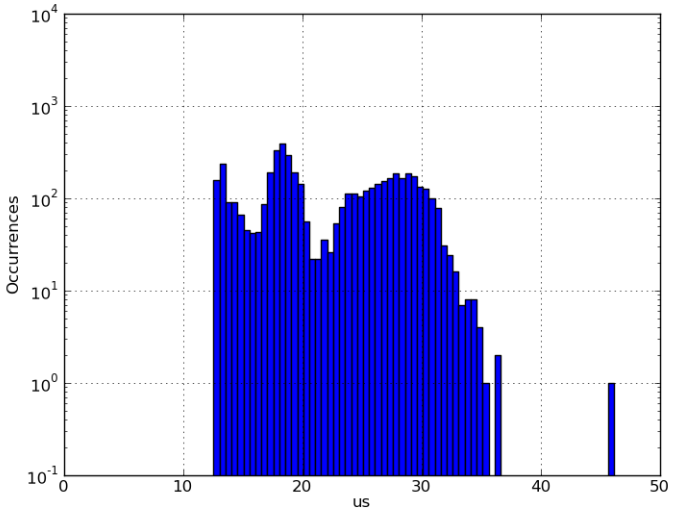
Figure 4.5. PREEMPT-RT M-Blocking SMP Load

9 values are actually modified by the correction, the resulting distribution is quite similar visually.

Clearly, the presence of competing load in a system with two CPUs has significantly modified the measured behavior but a particularly interesting aspect is that most of the increased variation in behavior is not a result of interrupt or preemption during the locking operation because the correction for these intervals has not significantly changed the character of the distribution. It is possible that contention to access physical memory by the two CPUs or bus contention for access to the disk or by the disk for DMA have contributed to spreading the distribution of m-blocking times. However, our current experimental framework does not permit investigation of such effects because they occur within the hardware and provide no software in which to place Datastream events.

Under load, PMGT-PI also produces results with a greater variance than the results for an unloaded system. Figure 4.6 depicts the results. The average for this data-set is 22.36 microseconds which is significantly higher than the average of 14.16 for the histogram in Figure 4.5 for PREEMPT-RT with the same load. There is about an 8 microsecond difference in the averages which is significantly larger than the 5 microsecond difference between the

two configurations on an idle system. This data-set contains 12 instances of interrupts and preemption with seven of these values being overflow values. Removing interrupt and preemption activity reduces the overflow and outlier values to yield a maximum value of 40.26 instead of 154.17. Additionally, the average remains stable at 22 microseconds but the standard deviation is reduced to 5.77 from 6.53. Clearly, the small number of interrupts suggests that the spread of the distribution, compared to PMGT-PI on an idle system shown in Figure 4.3, is not a result of preempt and interrupt activity that we measure and it seems reasonable to believe that the same influences that caused greater variance in the PREEMPT-RT data are at work in this case and are associated with the higher level of concurrent activity on the dual CPU machine. It is interesting to note that the distribution appears to have three peaks, even though the reason for the segregation of behavior is not clear.

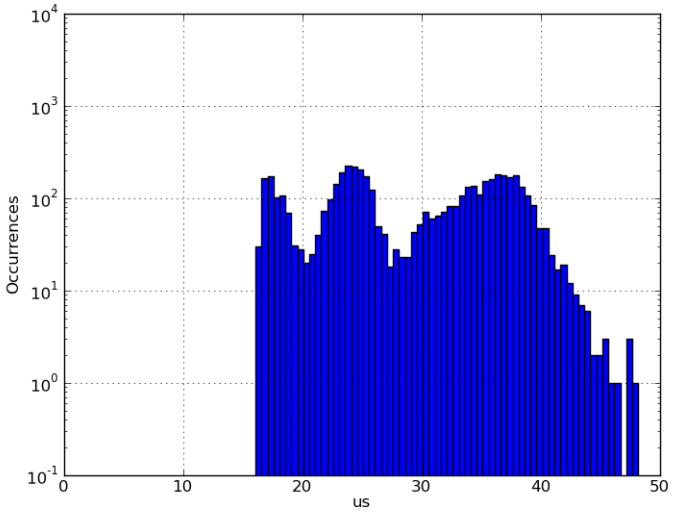


Count	5000
Buckets	100
Bucket Size	0.50
Min	12.67
Max	154.17
Underflow	0
Overflow	7
Average	22.36
σ	6.53
Median	21.46

Figure 4.6. PMGT-PI M-Blocking SMP Load

Figure 4.7 depicts the results of running the experiment on a dual CPU system with competing load under the HGS configuration. The average of 30.23 is about 8 microseconds larger than the average of 22.36 for PMGT-PI under the same conditions. This is almost double the 4.5 microsecond difference between HGS and PMGT-PI on an idle system. There

were 19 data values affected by interrupts and 9 of these values were overflow values. When the interrupts and preemption are removed the maximum for the distribution is reduced to 64.88 from 3098.72. The average is reduced slightly to 29.23 and the standard deviation is reduced to 7.44 from 46.67. However, the resulting distribution is still very similar to the original since fewer than 1% of the values were modified. As with PREEMPT-RT and PMGT-PI, the reason for the spreading of the distribution lies outside the preemption and interrupt effects currently being measured. The three peaks observed in the PMGT-PI distribution are also present.



Count	5000
Buckets	100
Bucket Size	0.50
Min	16.09
Max	3098.72
Underflow	0
Overflow	9
Average	30.23
σ	46.67
Median	29.89

Figure 4.7. HGS M-Blocking SMP Load

4.1.1.3 Uniprocessor Load

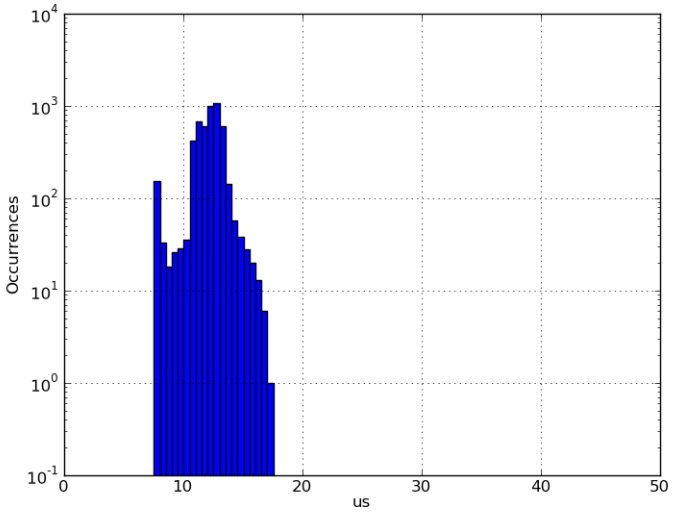
The gathered distributions for the SMP experiments with load had a larger than expected variance. Further, it was determined that the increased variance was not due to interrupts because few interrupts occurred as determined by post-processing. It is important to note that preemption in this context cannot occur unless an interrupt first occurs and so the absence of interrupts also implies the absence of preemption. One potential cause for the greater than expected variation in behavior is contention between the two CPUs due to

physical concurrency control used internally by the mutex implementation. In order to investigate that possibility, the experiments were repeated using uniprocessor configurations because physical concurrency control is not present in uniprocessor configurations. Note, however, that while contention among CPUs is eliminated by this configuration, contention for access to memory by the CPU and any DMA devices is still present.

Figure 4.8 depicts the distribution of m-blocking times that results from a uniprocessor PREEMPT-RT configuration under load. Visually, this histogram appears to have a smaller variance than the distribution for an SMP system with load depicted in Figure 4.5. This difference suggest that physical contention, at least in part for spinlocks, influenced the SMP experiment. However, this histogram still indicates more variance than Figure 4.2, so clearly there is another influence. The average of 12.34 microseconds for this histogram is higher than the average of 8.47 on an otherwise idle system and lower than average of 14.16 on an SMP system with load.

The standard deviation on the otherwise idle SMP system was 0.42 microseconds which rose to 4.04 with load. On a uniprocessor system under load, the standard deviation rose to 5.31. This rise in the standard deviation is unexpected and especially so given the appearance that Figure 4.8 is a tighter distribution than Figure 4.5. However, these standard deviation values are not completely representative of the histograms visually because they are greatly influenced by the presence of large outlier values that result from interrupts and preemption. Therefore, these histograms are better represented by considering the standard deviations of the distributions with the interrupt and preemption intervals removed. Under load, the adjusted SMP distribution has a standard deviation of 2.52 compared to the unadjusted 4.04. There were 12 data values effected by interrupts in the uniprocessor distribution and when this activity is removed the standard deviation is reduced from 5.31 to 1.36. The adjusted standard deviations of 2.52 for the SMP system and 1.36 for the uniprocessor system under load are far more in keeping with the visible shape of the histograms.

Figure 4.9 depicts the uniprocessor results for the PMGT-PI configuration under load.

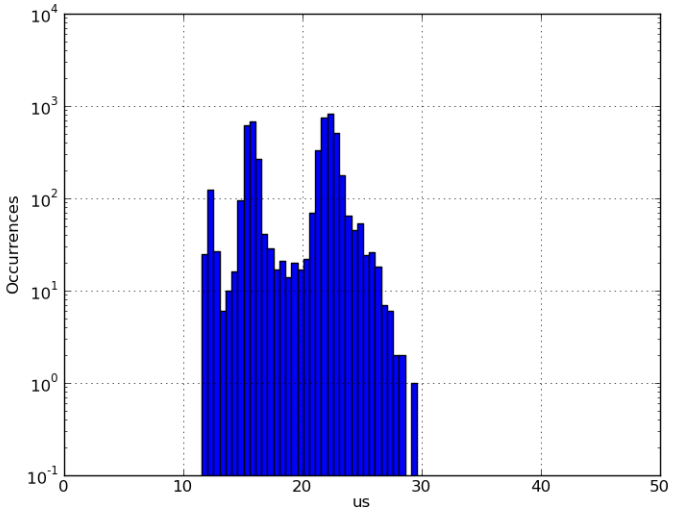


Count	5000
Buckets	100
Bucket Size	0.50
Min	7.54
Max	186.65
Underflow	0
Overflow	12
Average	12.34
σ	5.31
Median	12.29

Figure 4.8. PREEMPT-RT M-Blocking Uniprocessor Load

Again, this distribution appears to have a smaller variance than the SMP distribution in Figure 4.6 but it also seems to have a higher variance than the idle distribution in Figure 4.3. The average of 20.78 microseconds is higher than the average of 13.56 for an otherwise idle system and lower than the average of 22.36 for a loaded SMP system. This distribution contains 31 values effected by interrupts, of which all are overflow values, and when these data points are adjusted to remove the effects of the interrupts the maximum value for the distribution is reduced to 39.72 from 634.72. Additionally, the standard deviation is reduced from 18.59 to 3.68 and the average is reduced from 20.78 to 19.65. Again, as with the PREEMPT-RT results, this histogram is better represented by computing the standard deviation after the interrupt and preemption activity has been removed. This histogram has a lower adjusted standard deviation of 3.68 than the 5.77 microsecond adjusted standard deviation for a SMP system under load and a higher standard deviation than 0.90 for an otherwise idle system. From these adjusted numbers and from looking at the histograms it is apparent that running on a uniprocessor system has eliminated one source of variation in behavior compared to the unloaded system but others clearly remain. The three peaks for Figure 4.9 are particularly clear indicating at least two significant sources of variation in

behavior.

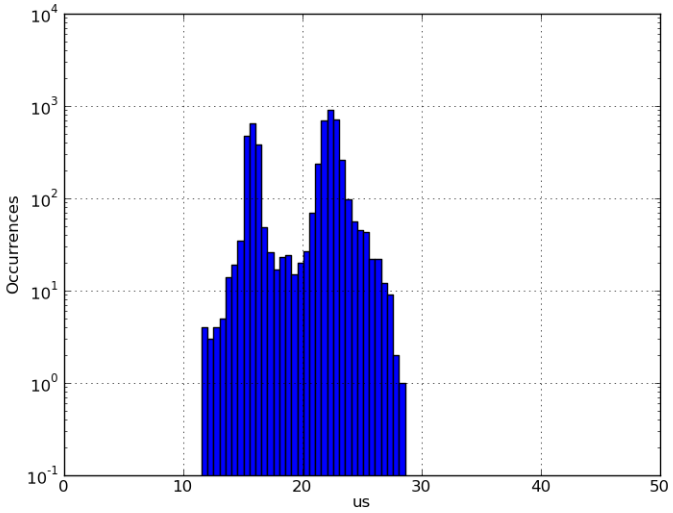


Count	5000
Buckets	100
Bucket Size	0.50
Min	11.90
Max	634.72
Underflow	0
Overflow	31
Average	20.78
σ	18.59
Median	21.61

Figure 4.9. PMGT-PI M-Blocking Uniprocessor Load Interrupt-able

The PMGT-PI uniprocessor configuration was also examined with interrupts disabled during proxy searches to investigate whether there is a source of interrupts that are not recorded by Datastream events. Figure 4.10 depicts the results. This histogram is similar to the interrupt-able histogram in several ways: they both have a minimum around 12 microseconds, an average of about 20, and a median of about 21. Visually, there are fewer values clustered around the minimum value but otherwise the histograms appear quite similar. Most importantly, the variance of 3.30 is near the adjusted variance of 3.68 for the interrupt-able histogram. This suggests that unaccounted for interrupts are not the cause of the variance of the interrupt-able histogram in Figure 4.9.

Figure 4.11 depicts the results for the uniprocessor HGS configuration. Like the other uniprocessor configurations, the variance of this histogram appears to be less than the variance of the distribution for a SMP system under load, depicted by Figure 4.7, and greater than the distribution for an idle system, depicted by Figure 4.4. The average of 26.45 microseconds is higher than the average of 17.93 for an otherwise idle system and lower than the average of 30.23 for an SMP system under load. This distribution contains 31 values



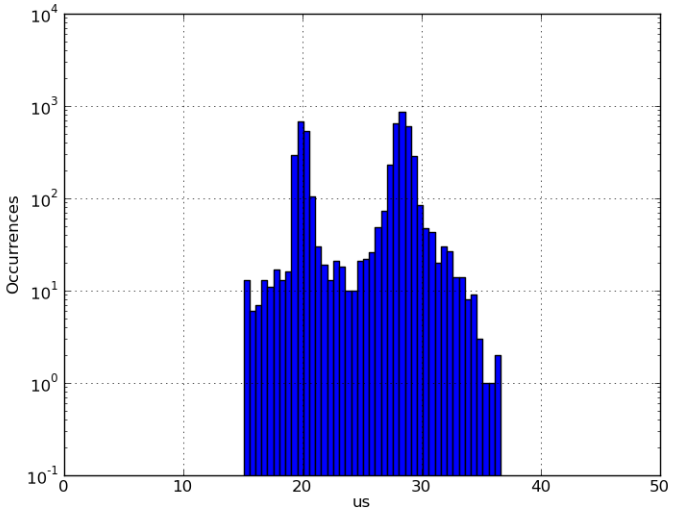
Count	5000
Buckets	100
Bucket Size	0.50
Min	11.86
Max	28.18
Underflow	0
Overflow	0
Average	20.19
σ	3.30
Median	21.90

Figure 4.10. PMGT-PI M-Blocking Uniprocessor Load Uninterrupt-able

influenced by interrupts, all of which are overflows, and when this influence is removed the maximum decreases to 48.91 from 707.76. The standard deviation is reduced from 18.80 to 4.35 and the average is reduced from 26.45 to 25.43. The adjusted standard deviation of 4.35 is lower than the adjusted standard deviation of 7.44 for an SMP system under load which indicates that a source of variance has been removed. However, since the adjusted standard deviation is larger than the 0.42 for an idle system there is still an unaccounted for source of variance.

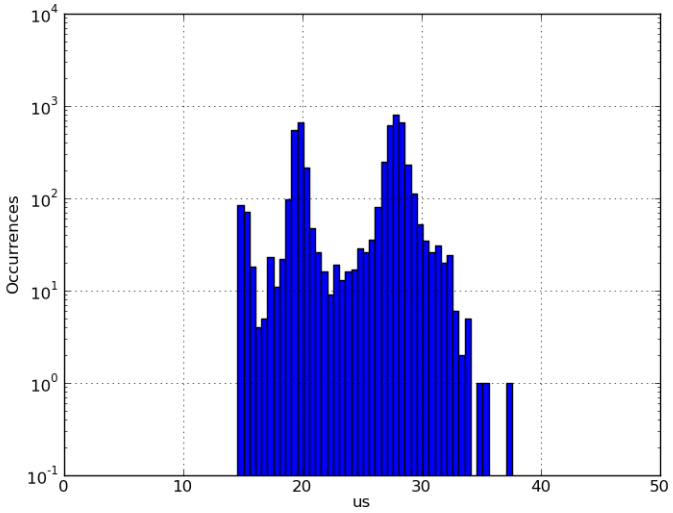
The results for HGS with interrupts disabled during proxy searches are depicted by Figure 4.12. The standard deviation of 4.42 microseconds for this histogram is close to the adjusted standard deviation of 4.35 for the interrupt-able histogram. There is about a 1 microsecond difference between the average of 24.65 for this histogram and the average of 25.43 for the interrupt-able histogram. Though there are a few differences, this histogram looks essentially identical to the interrupt-able histogram and, thus, it is unlikely that a significant portion of the variation in behavior of the uniprocessor system under load is due to an unaccounted for source of interrupts.

Overall, PMGT-PI has been shown to add 5 to 8 microseconds of overhead compared



Count	5000
Buckets	100
Bucket Size	0.50
Min	15.08
Max	707.76
Underflow	0
Overflow	31
Average	26.45
σ	18.80
Median	27.81

Figure 4.11. HGS M-Blocking Uniprocessor Load Interrupt-able



Count	5000
Buckets	100
Bucket Size	0.50
Min	14.63
Max	37.22
Underflow	0
Overflow	0
Average	24.65
σ	4.42
Median	27.24

Figure 4.12. HGS M-Blocking Uniprocessor Load

to the PREEMPT-RT base configuration and HGS has been shown to add from 6 to 8 microseconds of overhead in addition to the PMGT-PI overhead. When competing load was added, the distributions for PREEMPT-RT, PMGT-PI, and HGS experienced increased variance and currently this effect cannot be fully explained by the current experimental framework. However, since the effect was also present in PREEMPT-RT, it does not suggest that the effect is due to a property of the PMGT or HGS frameworks.

4.1.2 Unlocking Time

The experiment discussed in this section measures the time it takes the owner of a mutex to unlock a mutex with a single direct waiter and two indirect waiters. Unlocking a mutex involves picking the best waiter, designating it as the pending owner, and waking it up. The scenario executed using Guided Execution for this experiment is depicted by Figure 4.13. In this scenario, task $T1$ is unlocking the mutex $M1$ and task $T2$ is selected to become the pending owner. This scenario is executed 5000 times for PREEMPT-RT, PMGT-PI, and HGS to evaluate the difference in overhead between these implementations. No load was placed on the system during the experiment.

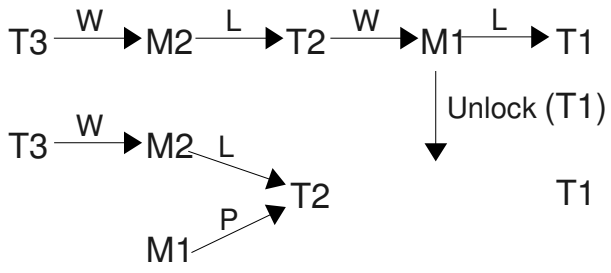


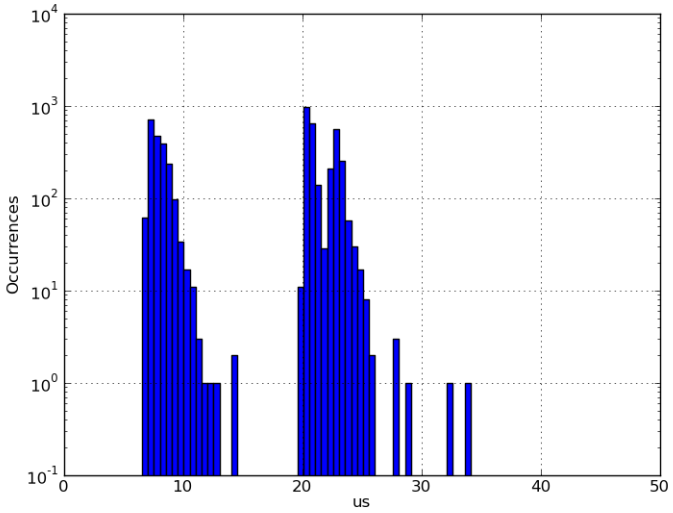
Figure 4.13. An Unlocking Scenario

The data for this experiment is gathered by emitting a Datastream event when $T1$ begins the unlock operation and a second Datastream event is emitted when the operation completes. Additionally, Datastream events are gathered that record the context switch and interrupt activity of $T1$ in order to determine if $T1$ is preempted or interrupted while it is performing the unlock operation. In post-processing, the elapsed time between the start and stop events is determined by taking the difference between their time stamps and that datum was inserted into a histogram.

In general, the unlock operation is not preemptable or interrupt-able but there is a brief period between the Datastream events and the unlock activity being measured during which preemption might occur. In this experiment we would thus subtract the preemption time

or eliminate the datum completely because it is an artifact of the measurement method and not a property of the algorithm.

Figure 4.14 shows the distribution of unlocking times for PREEMPT-RT. The high standard deviation of 6.96 for this histogram indicates that there is a major source of variance, probably resulting from interrupt and preemption. The histogram has two clearly defined clusters of data at 10 and 25. Examination of the event data reveals that all elements of the cluster at 25 experience an interrupt or preemption.

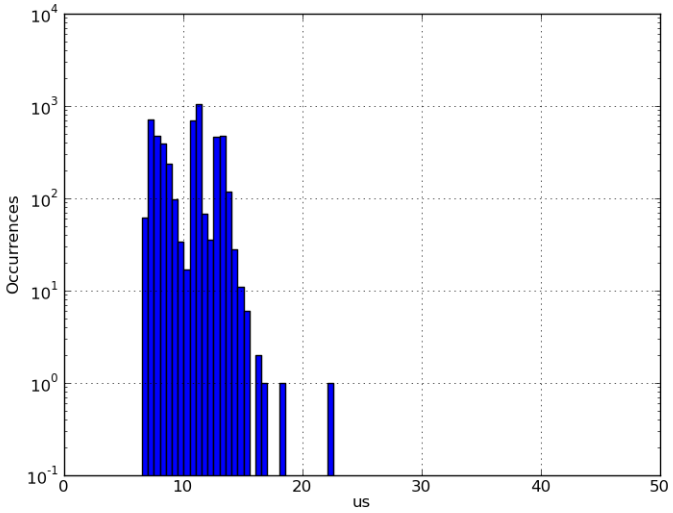


Count	5000
Buckets	100
Bucket Size	0.50
Min	6.74
Max	125.49
Underflow	0
Overflow	1
Average	15.98
σ	6.96
Median	20.36

Figure 4.14. PREEMPT-RT Unlocking

Figure 4.15 is the result of removing the interrupts and preemption intervals from the data in the cluster at 25 in Figure 4.14. The data in this figure is much more tightly grouped around an average of 10.29 with a standard deviation of 2.19. The maximum value is 22.22. Visually, the upper peak has almost merged with the lower but still seems discernible. We believe this is because subtracting the length of the interrupt service or preemption intervals leaves a small amount of preemption and interrupt overhead unmeasured.

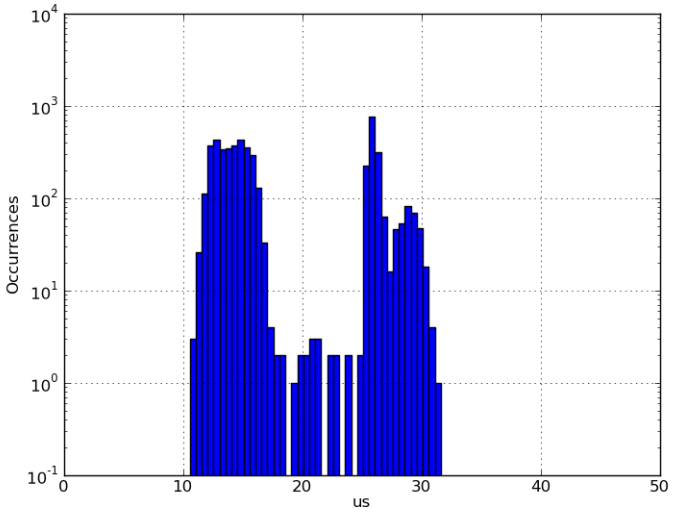
The results for the PMGT-PI configuration, illustrated by Figure 4.16, also have a large standard deviation of 6.07 and a bi-modal distribution resulting from interrupts and preemption. Figure 4.17 results from removing the interrupts and preemption from this data. The



Count	5000
Buckets	100
Bucket Size	0.50
Min	6.74
Max	22.22
Underflow	0
Overflow	0
Average	10.29
σ	2.19
Median	10.98

Figure 4.15. PREEMPT-RT Derived Unlocking

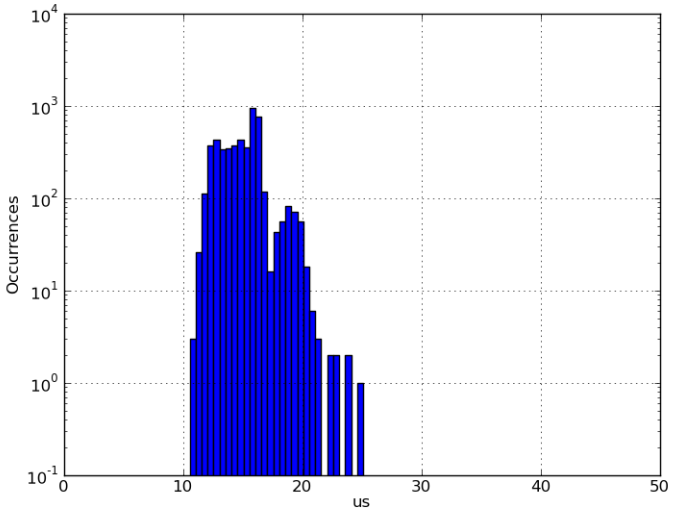
average of 14.94 is about 5 microseconds higher than the average of 10.29 for the PREEMPT-RT configuration indicating the increased overhead of PMGT. Visually, the upper peak has again essentially merged with the lower peak indicating that the majority of the distinction was due to the interrupts and preemption.



Count	5000
Buckets	100
Bucket Size	0.50
Min	10.89
Max	31.56
Underflow	0
Overflow	0
Average	18.32
σ	6.07
Median	15.11

Figure 4.16. PMGT-PI Unlocking

Figure 4.18 depicts the data for the HGS configuration. Once again, this configuration also has a high standard deviation of 6.71 which indicates that there are interrupts and



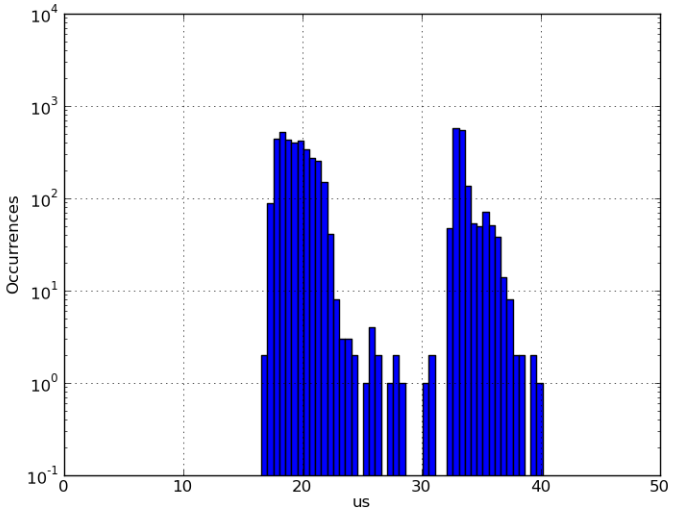
Count	5000
Buckets	100
Bucket Size	0.50
Min	10.89
Max	24.63
Underflow	0
Overflow	0
Average	14.94
σ	1.82
Median	15.11

Figure 4.17. PMGT-PI Derived Unlocking

preemption occurring. Figure 4.19 illustrates the HGS configuration data with the interrupts and preemption removed. Again, the upper peak largely merges with the lower indicating that essentially all of the bi-modal distribution is a result of interrupts and preemption. However, recall that the HGS support routines must do more complex concurrency control than the PMGT-PI versions and so some of the increase in variance may be due to the difference in concurrency control. The maximum for this configuration is the highest at 38.33. The average for this configuration is also the highest at 20.67 indicating the increase in overhead of the HGS scheduling layer compared to the Linux Scheduling Stack.

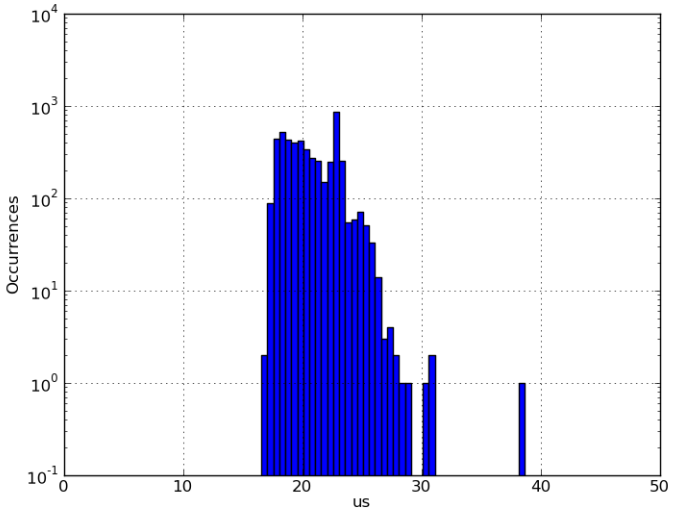
4.2 Scheduling Overhead

Scheduling time overhead was examined using two experiments that make use of pipeline based computations. Pipeline computations were used because they were readily available and they have been used with HGS in the past. The first experiment compares PREEMPT-RT to a simple HGS hierarchy and the second experiment examines a more complex HGS hierarchy under uniprocessor and SMP configurations. The scheduling time of the PMGT-PI configuration is not of interest because this configuration does not modify the Linux



Count	5000
Buckets	100
Bucket Size	0.50
Min	16.94
Max	39.63
Underflow	0
Overflow	0
Average	24.01
σ	6.71
Median	20.33

Figure 4.18. HGS Unlocking



Count	5000
Buckets	100
Bucket Size	0.50
Min	16.94
Max	38.33
Underflow	0
Overflow	0
Average	20.67
σ	2.15
Median	20.33

Figure 4.19. HGS Derived Unlocking

Scheduling Stack present in PREEMPT-RT and, thus, it has the same scheduling time overhead as PREEMPT-RT.

The metric for both of these experiments is the time that the system takes to make a decision at scheduling time. In the case of PREEMPT-RT this is the time required to evaluate the Linux Scheduling Stack. For HGS, the scheduling time is the amount of time required to evaluate the HGS hierarchy and, if no decision is made by the hierarchy, to

evaluate the Linux Scheduling Stack.

The scheduling time is measured by recording a time-stamp in a local variable immediately before a new task is picked and subtracting that time-stamp from a second time-stamp computed after a task has been picked. The resulting difference is stored in a Datastreams histogram which records the cumulative data for the entire experiment. The histogram is output at the end of the experiment. Gathering a histogram directly instead of deriving it from a set of events for each scheduling decision significantly reduces the instrumentation effect.

4.2.1 Simple Hierarchy

Scheduling time overhead was first measured using a pipeline consisting of five threads connected by AF_UNIX local socket pairs. These threads consist of a thread that receives messages, three threads that perform work, and a thread that acknowledges that the message has reached the end of the pipeline. A sixth thread, in a separate process, generates and sends messages to the pipeline using a loop-back AF_INET socket. Threads closer to the end of the pipeline are given a better priority. Under PREEMPT-RT the **nice** system call is used to set priorities of the pipeline stages relative to the base priority.

There are 10000 messages sent down the pipeline. Messages are sent at random intervals generated by a normal distribution with a mean of 5 milliseconds and a variance of 20 milliseconds. Each worker thread of the pipeline performs between 6000 and 8000 integer operations per message to represent work done processing the message. The number of operations is randomly generated using a uniform distribution. The pipeline is executed twice: once using PREEMPT-RT and once using a simple HGS hierarchy, on a dual CPU 996 MHz Pentium 3.

The HGS hierarchy used to control the pipeline for this experiment is depicted in Figure 4.20. This hierarchy contains a System group at the top that is controlled by a static priority scheduler. The System group checks its members in descending order of priority which is

from left to right in the figure. In this case, the pipeline application is the only one under HGS control so it is consulted first. If the pipeline application does not choose a thread then the System group gives the rest of the Linux Scheduling Stack control.

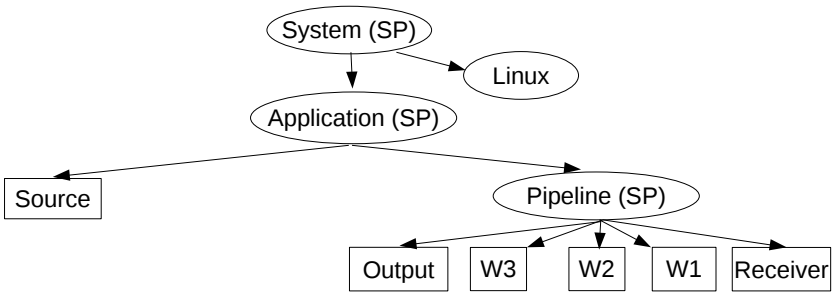
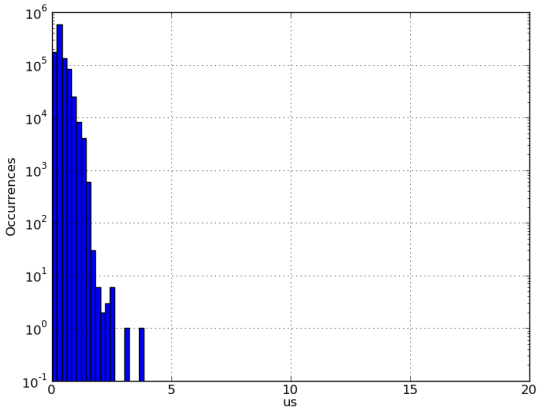


Figure 4.20. Pipeline HGS Hierarchy

The Application group has only two members controlled under a static priority scheduler. The top priority item is the message source and the other item is the group representing the pipeline. Obviously, this implements a greedy source of messages but could be modified if different message generation behavior was desired. For example, periodic message generation with specified variance could be used. The Pipeline group also uses static priority with the best priority at the end of the pipeline and the worst at the beginning. We refer to these as "drain semantics" because it encourages messages to move through the pipeline instead of buffering at the beginning of the pipeline. If we assigned priorities in the other direction then the first stage of the pipeline would tend to accumulate messages without passing them on.

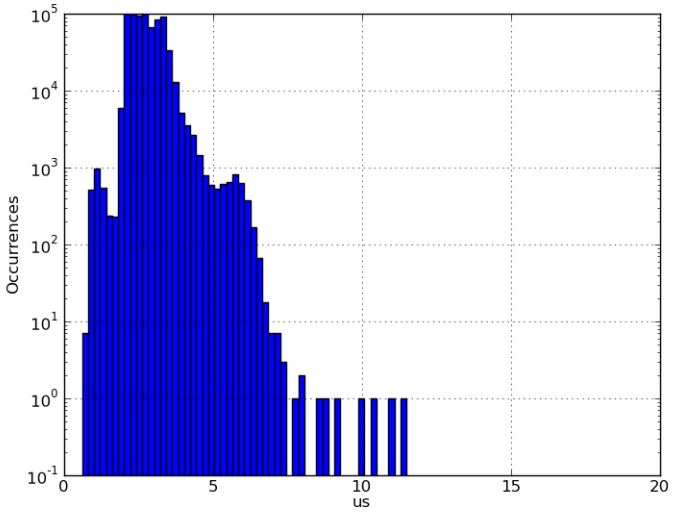
Figure 4.21 shows the results for PREEMPT-RT and Figure 4.22 shows the results for HGS. The 2.78 microseconds average value for HGS is somewhat higher than the 0.34 average for PREEMPT-RT in absolute terms since the increase is about 2.5 microsecond. In proportional terms, it is nine times the size which is obviously a significant increase. Part of this overhead occurs because HGS makes more decisions at scheduling time than PREEMPT-RT. Note however that PREEMPT-RT incurs overhead on every system timer tick to update

scheduling state which HGS does not. Obviously, timer ticks happen much less frequently than scheduling invocations but these factors complicate the question of comparing the overheads of the two approaches. The total number of scheduler decisions also differs. Under PREEMPT-RT there are slightly over one million invocations of the scheduler and under HGS there are fewer than 75% as many. Thus, while the individual decisions by HGS are more expensive, there are also fewer of them, further complicating the comparison.



Count	1014377
Buckets	100
Bucket Size	0.20
Min	0.09
Max	3.71
Underflow	0
Overflow	0
Average	0.34
$\sim \sigma$	0.21
$\sim$Median	0.30

Figure 4.21. Pipeline PREEMPT-RT Scheduler Overhead



Count	728316
Buckets	100
Bucket Size	0.20
Min	0.73
Max	11.39
Underflow	0
Overflow	0
Average	2.78
$\sim \sigma$	0.55
$\sim$Median	2.72

Figure 4.22. Pipeline HGS Scheduler Overhead

PREEMPT-RT has a standard deviation of 0.21 and HGS has a standard deviation of

0.55. This suggests that HGS is slightly more variable than PREEMPT-RT in absolute terms, but it is interesting to note that when looking at the standard deviation as a fraction of the average value then HGS is less variable than PREEMPT-RT. However, the larger absolute deviation is expected since HGS sometimes consults the hierarchy and the Linux Scheduling Stack instead of only consulting the Linux Scheduling Stack as in PREEMPT-RT.

The maximum value of 11.39 for HGS is much larger than the maximum value of 3.71 for PREEMPT-RT. This large increase may be due to multiple invocations of the static priority scheduler associated with the Application and Pipeline groups due to the details of how proxies are used. Recall that HGS schedulers can select tasks that are m-blocked to cause the proxy for the task to run. However, if the proxy is blocked then the scheduler is asked for another selection. Additionally, the static priority scheduler doesn't monitor when a task is blocked for reasons other than a mutex. For these reasons, this scheduler can, in theory, make several unusable selections before picking a task that can actually be run. Since a significant amount of I/O is performed by this experiment, it is likely that some of the tasks are blocked due to I/O, which would make the probability of selecting a task that is blocked or has a blocked proxy somewhat greater. Under PREEMPT-RT, there are fewer reasons for the decision time to vary because tasks that are not runnable can never be selected and, since PREEMPT-RT does not use proxy relations at scheduling time, a blocked proxy will never be considered.

4.2.2 Complex Hierarchy

Scheduling time overhead was also measured using a more complex HGS hierarchy. The experiment presented in this section uses four pipelines with the same structure as the pipeline in Section 4.2.1 but the parameters controlling message generation and the amount of work performed are somewhat different.

Figure 4.23 depicts the hierarchy used to control the pipelines. The goal of this hierarchy is to balance the progress of the pipelines. The Application group is a static priority group

that controls the application. The Senders group is a round robin group that alternates between the different sources for the pipelines. This group is always consulted before the Pipelines group which controls the pipelines, thus giving preference to the message sources which insures that all messages which should be generated by the statistical source model are available to the proper pipeline. The Pipelines group uses a Frame Progress (FP) scheduler that balances the progress of the pipelines based on the progress reported by each of the pipelines. Each pipeline is controlled by a static priority group that gives later stages of the pipeline a better priority. The Output thread of each pipeline informs the Frame Progress scheduler about the progress of the pipeline in the form of a last-frame-processed state variable. The worker threads, denoted by a name starting with"W", perform work on the messages. Finally, the Receiver thread for each pipeline receives messages from the source and sends them to the worker threads.

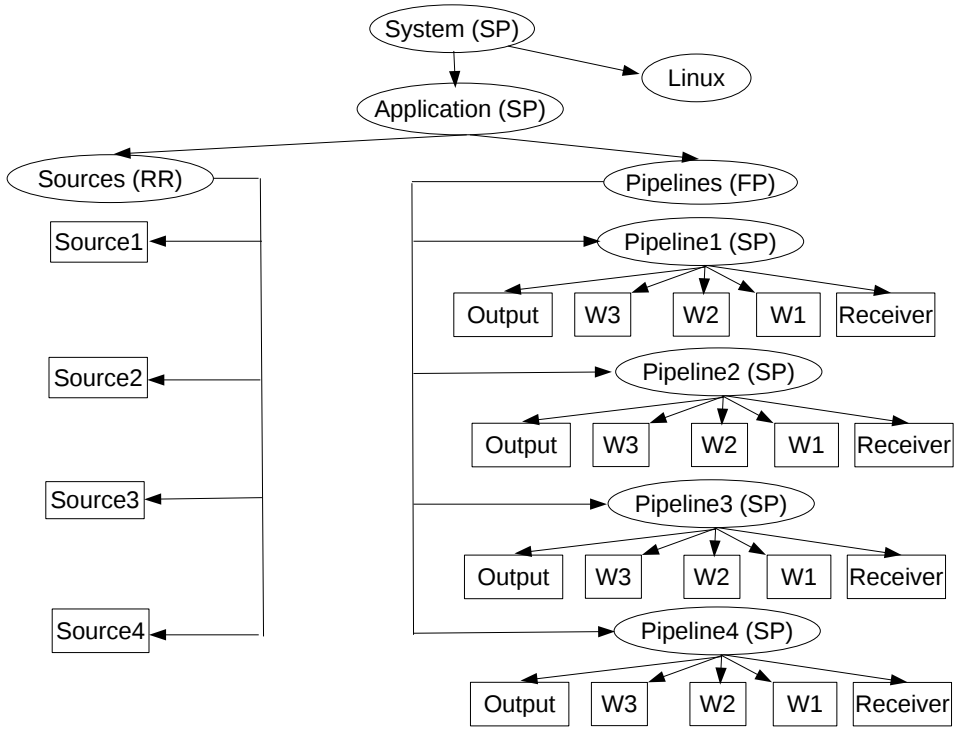


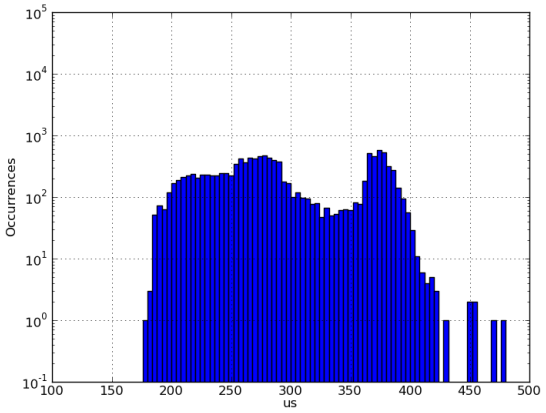
Figure 4.23. FP HGS Hierarchy

The amount of work performed for each message in a pipeline is generated using a uniform distribution. Pipeline1 sends messages every 33 milliseconds with a variance of 1 drawn from a normal distribution, while each worker thread performs between 5000 and 7000 integer operations for each message. Pipeline2 sends messages at an interval based on a normal distribution with a mean of 5 milliseconds and a variance of 20 milliseconds. Each worker thread performs between 6000 and 8000 integer operations for each message. Pipeline3 sends messages at an interval based on a normal distribution with a mean of 20 milliseconds and a variance of 20 milliseconds. Each worker thread performs between 10000 and 10200 integer operations for each message. Pipeline4 sends messages at an interval of 33 milliseconds with a variance of 1 millisecond drawn from a normal distribution and performs between 5000 and 10000 integer operations per messages.

Figure 4.24 depicts the distribution of the amount of time spent processing each message by the worker pipeline stages. The figure indicates a significant range of processing times for worker threads which ensures that the scheduling overhead of the system is not affected by artificial synchronization among computation components that might be produced by similar processing times. This histogram has the expected 12000 samples that represent 12 worker threads processing 1000 messages. The 16 overflow values are a result of preemption during the processing of a message but these aren't particularly of interest since this experiment is primarily interested in scheduling time overhead.

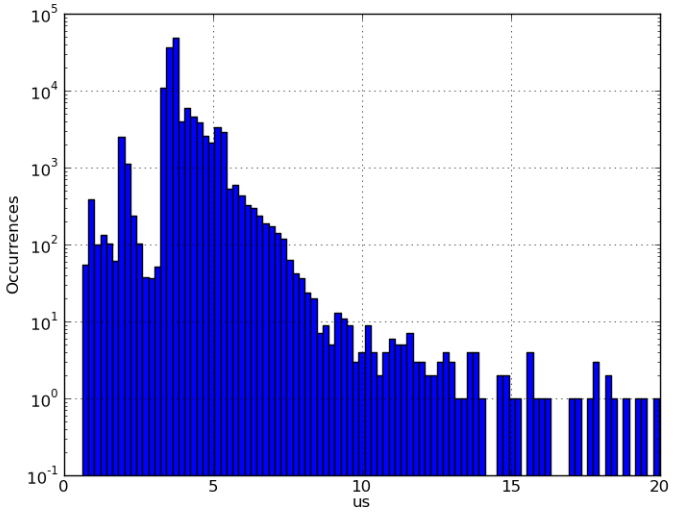
This experiment was executed twice, once on a one CPU 996 MHz Pentium 3 and once on a dual CPU 996 MHz Pentium 3. Figure 4.25 shows the results for the single CPU execution and Figure 4.26 shows the results for the two CPU execution.

The average of 5.04 microseconds for the SMP configuration is about 1 microsecond higher than the average of 3.82 for the uniprocessor configuration. The maximum value of 29.08 for the SMP configuration is significantly higher than the maximum value of 19.88 for the uniprocessor configuration. Currently, HGS constrains concurrency during hierarchy evaluation by permitting only one thread to execute within or under a given group at a time.



Count	12000
Buckets	100
Bucket Size	4.00
Min	179.82
Max	961618.98
Underflow	0
Overflow	16
Average	449.993
$\sim \sigma$	336.20
$\sim$Median	278.00

Figure 4.24. Message Processing Time

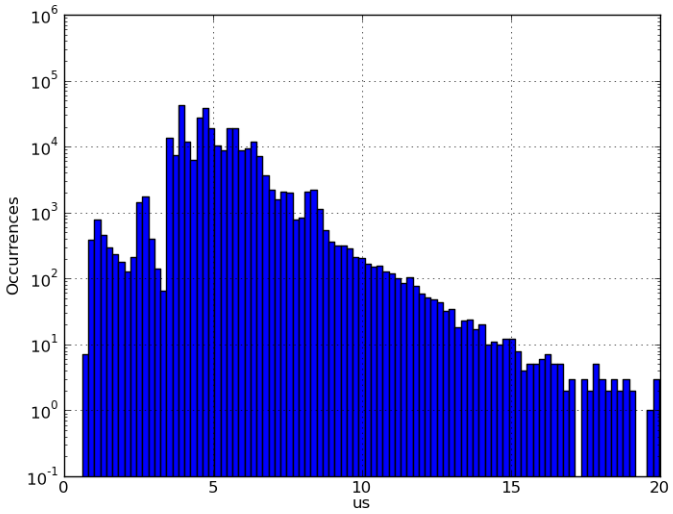


Count	133672
Buckets	100
Bucket Size	0.20
Min	0.61
Max	19.88
Underflow	0
Overflow	0
Average	3.82
$\sim \sigma$	0.76
$\sim$Median	3.73

Figure 4.25. FP UP HGS Scheduler Overhead

Future development of HGS should be able to relax this constraint. This constraint implies that a CPU might have to wait for another CPU to finish sub-hierarchy evaluation. The higher average and maximum values for the SMP configuration suggest that there may be some physical contention taking place between CPUs.

The standard deviation of 1.29 microseconds for the SMP configuration is substantially higher than the standard deviation of 0.76 for the uniprocessor experiment indicating that the values are more spread out. The figures indicate this visually in that the SMP histogram is much wider than the uniprocessor histogram but many of the buckets contain many fewer



Count	290724
Buckets	100
Bucket Size	0.20
Min	0.77
Max	29.08
Underflow	0
Overflow	26
Average	5.04
$\sim \sigma$	1.29
$\sim$Median	4.74

Figure 4.26. FP SMP HGS Scheduler Overhead

samples than the peaks. Recall that the vertical scale is exponential. The increased variance also produced 26 overflow values in the SMP histogram that are above 20. These overflows are not pictured to put the two histograms on the same scale for comparison purposes.

The SMP experiment had 290724 samples which is about twice as many as the uniprocessor experiment which had 133672 samples. This appears reasonable because a machine with two processors must make more scheduling decisions than a machine with a single processor that has similar behavior. However, the experiment is processing the same number of messages with the same set of threads and so the SMP configuration making over twice as many scheduling decisions during the period of the experiment is somewhat puzzling.

Finally, it is worth mentioning that the average of 5.04 microseconds for the SMP configuration and the average of 3.82 for the uniprocessor configuration are higher than the average of 2.78 for the simpler hierarchy in Section 4.2.1. Therefore, the overhead added by this hierarchy is about 1 to 2 microseconds. However, since it is controlling three times as many threads in a significantly more complex hierarchy, an increase of 2 microseconds is not particularly surprising.

4.3 Memory Overhead

PMGT explicitly represents waiting relations using a node structure. For long m-blocking chains, there may be many nodes because each waiter in the chain has a node for each link of the chain. Moreover, the fixed pool of nodes used by PMGT could be depleted if there are many long m-blocking chains. Therefore, the experiment presented in this section examines the node use on a heavily loaded system where contention for mutexes could be intense. Various aspects of node structure use are of interest including: the total number of nodes in use, the total number of waiters on a mutex, and the length of the m-blocking chains since each waiter has a node for each link in its m-blocking chain.

Five measurements were recorded by the memory overhead experiment: (1) the number of nodes allocated by PMGT, (2) the number of nodes associated with each mutex, (3) the number of tasks on the system, (4) the number of m-blocked tasks, and (5) the number of links in an m-blocking chain traversed by a task that searches for a proxy.

The number of nodes allocated by PMGT is tracked using an atomic counter that is incremented as nodes are allocated from the node pool and decremented when they are freed. The value of this counter is added as a sample to a histogram each time a mutex operation is performed.

A count of the nodes associated with a mutex is tracked using an atomic counter stored in the mutex data structure. This counter tracks the number of node structures used to represent the set of waiters on the mutex. The counter is incremented each time a node is associated with the mutex and decremented each time a node is removed from the mutex. The value of this counter is added as a sample to a histogram each time a mutex operation is performed.

The total number of tasks on the system is tracked using an atomic counter which is incremented each time a new task is forked and decremented each time a task is freed. The value of this counter is added as a sample to a histogram each time a task is forked.

The number of m-blocked tasks is tracked using an atomic counter which is incremented

shortly before a task blocks after searching for its proxy due to trying to acquire a mutex owned by a different task and decremented when the task wakes up. The value of this counter is added as a sample to a histogram each time a mutex operation is performed.

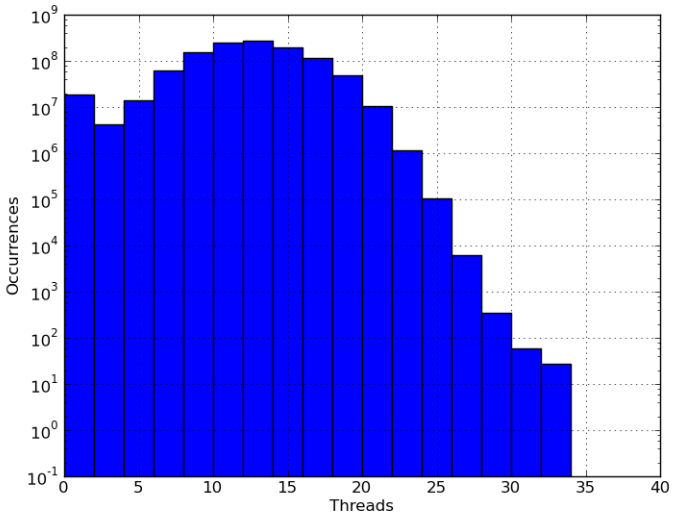
The number of links in an m-blocking chain searched by an m-blocking task is recorded by a local variable as it searches the chain for the proxy. The chain length is added as a sample to a histogram at the end of the search when the proxy is found.

All of the gathered histograms are output at the end of the experiment. The instrumentation effect is significantly reduced by directly gathering histograms rather than deriving them from a large number of events during post-processing.

The set of tasks in the experiment is generated by simultaneously compiling three Linux kernels on a KVM virtual machine configured with eight CPUs. The "-j6" option was specified for all of the kernel compiles in order to ensure that a plentiful number of active threads was available. The virtual machine ran on a server with eight 2.3 GHz Xeon CPUs and was configured to use all eight physical CPUs. This experiment was conducted using the PMGT-PI configuration. It was observed, by running "top" during the experiment, that CPU utilization for each CPU varied between 75% and 95% indicating that all of the CPUs for the system were kept reasonably busy and that contention over shared system mutexes was likely to occur.

Figure 4.27 shows the number of tasks that are m-blocked on the system. This histogram has over a billion samples because a sample is recorded during each mutex operation and mutexes are heavily used. Note that this counts the number of tasks in the m-blocked state but says nothing about how many tasks are waiting on a particular mutex. This histogram has an average of 11.95 and the data is fairly spread out as indicated by a standard deviation of 5. The highest values, such as the maximum bucket value of 32, make up a small percentage ($\tilde{1}$/100,000 of 1%) of the samples.

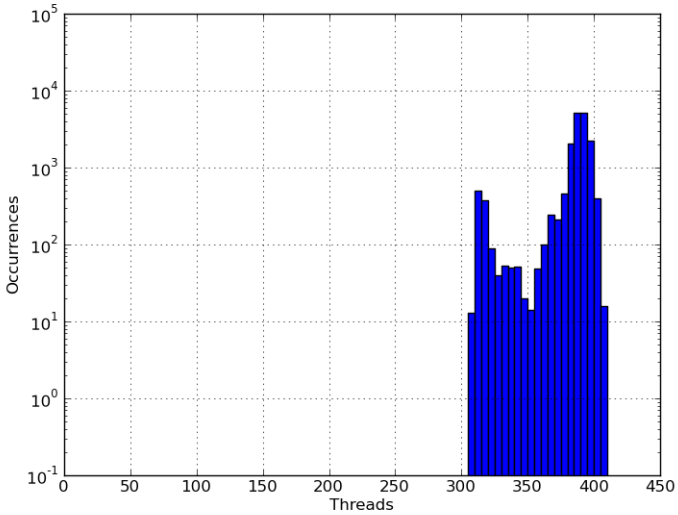
Figure 4.28 illustrates the total number of tasks on the system. The minimum value of 307 is the number of tasks present at the start of the experiment. The tasks present



Count	1.14×10^9
Buckets	20
Bucket Size	2
Min	0
Max	32
Underflow	0
Overflow	0
Average	11.95
$\sim \sigma$	5.00
$\sim$Median	13.00

Figure 4.27. M-Blocked Tasks

at the start of the experiment are the tasks that Linux uses to maintain the system. Up to a hundred additional tasks were active at sampling times during the experiment. Many of these tasks are a result of the kernel compilations generating hundreds of individual file compilation commands which each create several threads.

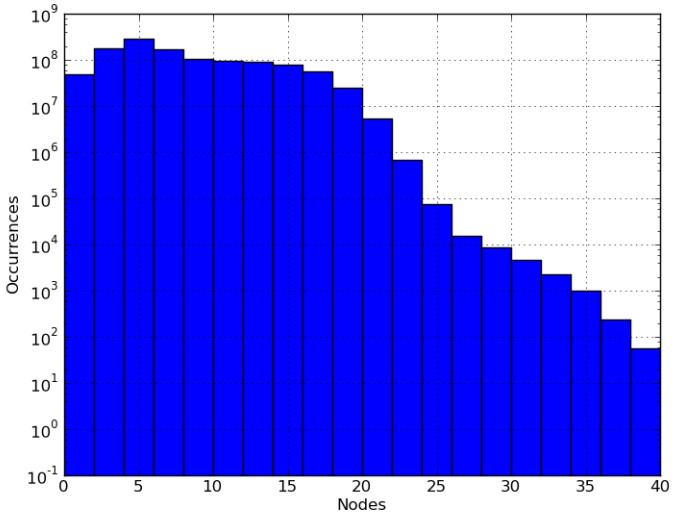


Count	17352
Buckets	100
Bucket Size	5
Min	307
Max	407
Underflow	0
Overflow	0
Average	383.85
$\sim \sigma$	27.60
$\sim$Median	387.50

Figure 4.28. Tasks

The total number of nodes used by PMGT is illustrated in Figure 4.29. The average value of 7.48 shows that usually node usage is quite modest. On a 32-bit system, a node has

a size of 68 bytes. Therefore, given the standard deviation of 5.48, the amount of memory used for nodes is usually between 136 bytes and 884 bytes. Memory usage is trivial even at the maximum value of 40 nodes which uses 2.6 KB of memory. Note that the maximum value is 40 and the last bucket contains values 38 and 39. Therefore, the 11 overflow values are all for a count of 40 nodes. However, we wanted to make other histograms use the 0 to 40 scale so accepting 11 overflow samples for this one histogram seemed reasonable.

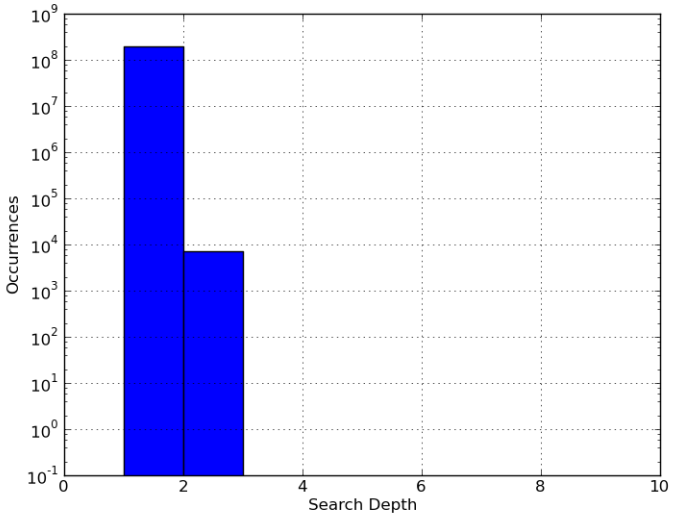


Count	1.14×10^9
Buckets	20
Bucket Size	2
Min	0
Max	40
Underflow	0
Overflow	11
Average	7.48
$\sim \sigma$	5.48
~Median	7

Figure 4.29. PMGT Nodes

This experiment also provides some clues about the structure of m-blocking chains that are created. Figure 4.30 depicts the number of links in an m-blocking chain traversed by a task that is searching for a proxy because it is locking a mutex owned by another task. Obviously, this histogram contains samples only gathered at locking time since this is the only time that this operation takes place. These results show that only searches of length one or two took place. In the best case, this means that chains of length two, containing at least three tasks and two mutexes, and chains of length one, containing at least two tasks and one mutex, were formed. It is possible that a longer chain of length four could form if a chain of length two joins another chain of length two. In this case, the m-blocking task would search two links of the chain it is joining. However, chains of this length are likely to

be rare and short lived because there are no records of the searches of length longer than 2 that would be required for a task to join a chain of length 4.



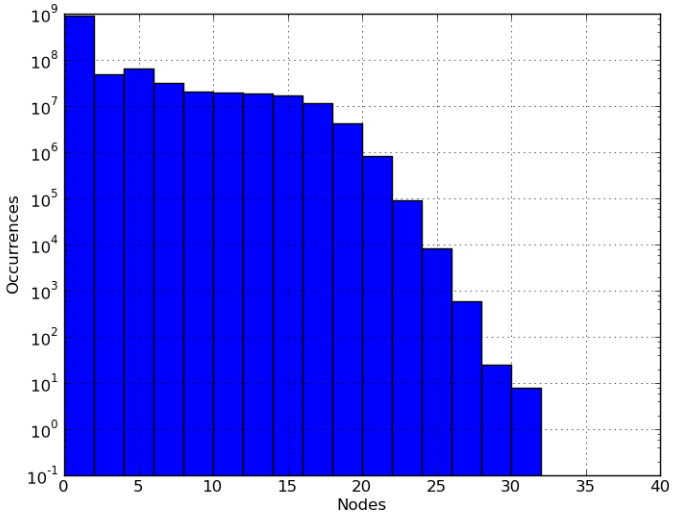
Count	1.14×10^9
Buckets	10
Bucket Size	1
Min	1
Max	2
Underflow	0
Overflow	0
Average	1
$\sim \sigma$	1.12
$\sim$Median	1

Figure 4.30. Search Depth

It is also worthwhile to note that this histogram contains only 195151916 samples out of the 1143527505 mutex operations performed. The primary reasons for this are: (1) only lock operations are relevant and so we would expect at most one half of the larger sample count and (2) only lock operations on a mutex that is already owned will walk the chain. So if the number of lock and unlock operations is essentially equal then there should be $1.14 \times 10^9/2 \approx 5.7 \times 10^8$ lock operations. Of these, approximately 3.7×10^8 were apparently on free mutexes since only 1.95×10^8 were samples in this histogram. Similarly, the total number of mutex operations on contested mutexes should be roughly twice the number of samples in the histogram or approximately 4×10^8. This is interesting because PMGT only performs accounting for contested mutexes.

Figure 4.31 shows the number of nodes associated with a mutex being operated on. The number of nodes associated with a mutex indicates the number of tasks that are directly and indirectly awaiting the mutex. The average of 1.55 indicates that the number of waiters is usually small, however, the earlier analysis also indicates that roughly 3.7×10^8 lock

operations were performed on free mutexes. In the histogram, the first bucket includes both values of 0 and 1 so we cannot tell how many of those samples were 0 and how many 1. In the worst case, a mutex has 31 nodes and, thus, 31 waiters. Given that Figure 4.30 shows that only short searches of chains are taking place, the m-blocking chain that this mutex is a part of is wider than it is long. A bushy tree of waiters is advantageous for PMGT because it requires fewer nodes for each waiter than a set of waiters with longer paths between waiters and the proxy.



Count	1.14×10^9
Buckets	20
Bucket Size	2
Min	0
Max	31
Underflow	0
Overflow	0
Average	1.55
$\sim \sigma$	3.98
~Median	1

Figure 4.31. PMGT Mutex Node Count

Most of the mutex operations associated with a kernel compile on a local disk are likely related to the file-system and all three simultaneous compiles for this experiment were on the local disk of the KVM guest OS. Most Linux file-systems use fine-grained concurrency control which uses separate mutexes for different resources. This type of concurrency control reduces contention. Therefore, in this case, short chains are to be expected. Coarse-grained concurrency control, which uses a single mutex for multiple resources, or a single resource with a lot of contention may produce longer m-blocking chains. However, particularly for real-time systems, this type of concurrency control is generally a bad design because many tasks may be m-blocked for a long period of time and this can produce undesirable system

behavior.

4.3.1 Summary

In summary, PMGT adds significant overhead to mutex operations and HGS demonstrates that scheduling layers may incur significant additional overhead when handling proxy relations. In compensation for the increased overhead, however , PMGT and HGS provide greater control over system behavior. Additionally, both PMGT and HGS have not yet been fully optimized.

HGS further demonstrates the price paid for greater control by adding modest scheduling overhead for small hierarchies and greater, more variable overhead for larger hierarchies. However, significant optimization of how a hierarchy is evaluated still needs to be performed.

The results in this section are all on the order of 10s of microseconds and, therefore, PMGT and HGS are probably not desirable for system semantics involving timescales smaller than hundreds of microseconds at present. In the future, PMGT and HGS could possibly be optimized to provide configurable system semantics using smaller time scales.

PMGT memory overhead is minimal when used with the fine-grain concurrency control of the Linux kernel because m-blocking chains tend to be wide and short. These types of chains require only a few node structures for each waiter. The results under the high load of 3 concurrent kernel compilations indicate that the memory overhead of this approach is extremely unlikely to be problematic under any approach. The memory overhead may increase for coarse-grained concurrency control, which might be utilized by user applications, but this type of concurrency control is often undesirable because it greatly constrains concurrency control.

Chapter 5

Conclusions and Future Work

Priority scheduling semantics are widely used but often it is difficult or complicated to map application semantics to priority semantics. Often, it is easier and less complicated to explicitly specify the semantics of the application to produce the desired application behavior. Several projects, such as Group Scheduling [1], Litmus-RT [4], and the Hierarchical Loadable Scheduler [16], have attempted to provide configurable scheduling semantics.

However, configuring the scheduling semantics of an application is not always sufficient to produce a desired application behavior if the application's concurrency control semantics are not also configured to produce that behavior. Most projects providing configurable scheduling semantics do not address configurable concurrency control semantics but there have been some efforts to implement concurrency control semantics that complement a particular set of scheduling semantics. For example, the Priority Inheritance implementation provided by the PREEMPT-RT patch complements the priority based scheduling semantics used by Linux [17].

The goal of the work presented in this thesis has been to demonstrate a system that allows complementary scheduling and concurrency control semantics to be configured to produce the desired application and system behavior. This goal is accomplished by integrating Proxy Management (PMGT) with the configured scheduling layer. PMGT provides an accounting

method independent of scheduling semantics used by mutex concurrency control and which supports the system scheduling layer with a configurable interface. It supports configuration of both how waiters interact with a proxy and how the next owner of a mutex is selected.

In the work described in this thesis, we have illustrated this approach with two scheduling layers: the priority based PREEMPT-RT semantics and the configurable Hierarchical Group Scheduling (HGS). HGS allows an application to specify its own scheduling and concurrency control semantics at runtime. Additionally, it provides a hierarchical organization of different applications to allow applications with different semantics to co-exist on the same system. HGS provides automatic handling of proxy relations that is applicable to a wide range of scheduling semantics.

The work presented in this thesis adapted HGS to coexist with priority-based Linux scheduling semantics. First, a Linux scheduling class was created to represent tasks controlled by HGS within the existing Linux scheduling framework. Second, HGS manages proxy relations between HGS tasks and Linux tasks to allow proper use of the proxy relation, regardless of the scheduling classes of the waiter and the proxy. Another significant contribution of this thesis is the extension of the basic handling of proxy relations to support multi-processor system configurations.

Guided Execution illustrates an example of a highly specialized programming model whose semantics can be directly implemented as a scheduler under HGS. This was used to implement a test suite for PMGT of 31 scenarios which cover all of the situations in which the mutex code can execute that we have generated through extensive review of all possible paths through the mutex code. Thus, we have considerable confidence in the completeness of the test suite but a more formalized and tool based approach to generating any missing scenarios is a reasonable part of future work.

A tool-set using configuration files was created that supports specifying each scenario and then automatically generating and executing the code for the scenario under the control of Guided Execution. The Guided Execution framework and scenario tool-set thus make it

possible to add any scenarios in the future which represent execution situations that have not yet been covered or which may be necessary due to changes in the mutex code. The 31 PMGT scenarios were then used to generate the HGS test suite which contains over 400 tests. Each scenario at the PMGT level becomes several tests which differ from each with respect to the scheduling classes, the CPU assignments, and the scheduling parameters of the tasks involved.

The evaluation presented in this thesis also investigated the overhead of the lock and unlock mutex operations, the overhead of making a scheduling decision, and the memory overhead of PMGT.

There is a moderate increase in overhead associated with performing proxy accounting during the lock and unlock mutex operations compared to the Priority Inheritance algorithm used by PREEMPT-RT. This increase in overhead has two components: that which is directly related to proxy accounting and that which is related to the notification of the scheduling layer by PMGT. On a loaded system, both the PREEMPT-RT and PMGT mutex operations displayed considerably more variance than on an idle system. All of the sources of this variance could not be accounted for using the experimental framework since significant variance remained both preemption and interrupts were eliminated from the mutex locking measurement period.

PMGT can be configured to use Priority Inheritance which does not increase scheduling decision overhead and it can also be configured to work with scheduling layers that make direct use of proxy relations to make a scheduling decision. HGS demonstrates a hierarchical scheduling layer which directly uses proxy relations. It was found that a small HGS hierarchy adds modest scheduling decision over Priority Inheritance. More complex hierarchies add additional overhead in exchange for precise control.

PMGT adds an explicit representation to Linux of the relations between tasks and mutexes for proxy accounting. This representation requires additional memory to be used by the system. However, the memory overhead of PMGT for the existing set of Linux mutexes

was found to be quite modest for a load consisting of multiple kernel compiles. Additional mutex use could increase the memory overhead but substantial increase would be necessary before memory use became a significant issue.

The work presented here has demonstrated that proxy accounting is a viable method of tracking waiting relations for kernel mutexes which can be configured to produce the currently hardwired semantics of PREEMPT-RT and which can be configured for other semantics as well. The overhead of this approach is not trivial and whether it is appropriate for a given system implementation or not will depend on the cost-benefit trade-off between the increased overhead and the ability to directly implement desired semantics. The use of Guided Execution to evaluate the correctness of the implementation also illustrated that some specialized semantics are not sensitive to the increase in overhead.

Much of the work presented in this thesis is focused on the configuration of scheduling and concurrency control semantics. This work completes the PMGT framework which makes mutex semantics both integrated with scheduling and considerably configurable but it does not directly address concurrency control at the user level. However, the extension of configurable concurrency control semantics to user-space mutexes, otherwise known as futexes, is among the most interesting and near term extensions of the work presented here.

A user-level futex could be associated with a system mutex when under contention and it would be useful for specialized applications to be able to specify what the mutex policy of the system mutex which is associated with a futex should be. This would allow applications to configure the concurrency control semantics of the system-level mutex representing the user-level futex to compliment scheduling semantics specified using HGS. The existing support for system-level mutexes would thus provide the same support for user-level futexes if such a relationship was created between user-level mutexes and system-level mutexes.

Some portion of this approach already exists in the form of PI-futexes. Normal futexes in standard Linux communicate with the kernel when the futex is under contention but the futex support is completely independent of OS level concurrency control. The PI-futex instead

uses a system level futex to represent the user-level futex under contention, thus gaining the priority inheritance capabilities. At the system level, this is essentially what would be required for user-level futexes with configurable semantics and integration with HGS. However, creating a user-level interface with sufficient power would still require additional work.

The user-level interface for configuration could be accomplished by creating a system call that records the desired mutex policy for a specific futex and then modifying the futex implementation to associate that policy with the appropriate system mutex when the system mutex is associated with the futex. Additionally, a method for specifying an application specific mutex policy at runtime would be needed. Obviously, the application specific policies would have to be implemented and loaded into the kernel as modules, much like HGS application specific schedulers.

There are several performance improvements for PMGT and HGS which could also be made. PMGT might be re-organized to manage its representation of m-blocking chains differently. When a task tries to acquire an already owned mutex it currently searches for its proxy by traversing the m-blocking chain in order to set up waiting relations with other mutexes in the m-blocking chain. However, the proxy for the task is obvious because it is also the proxy for the owner of the mutex. If the PMGT implementation can be re-organized to use data structures representing relations among waiters and proxies in a more localized way then overhead of walking the m-blocking chains and acquiring and releasing locks associated with each step could be eliminated. In this approach, the same relations would be represented but the data structures would be managed in a way that concentrates on the proxy in its approach to concurrency control, thus considerably simplifying the concurrency control and presumably reducing its overhead.

Furthermore, constraints currently placed on physical concurrency by HGS could be relaxed. One way to do this would be to transform the HGS group locks into only controlling the consistency of group level information and maximizing the use of per-CPU concurrency

control similar to that used by the Linux Scheduling Stack which maximizes physical concurrency. This would increase the physical concurrency of scheduling time decisions and make HGS more appropriate for systems with a large number of CPUs.

References

[1] Tejasvi Aswathanarayana, Douglas Niehaus, Venkita Subramonian, and Christopher Gill. Design and performance of configurable endsystem scheduling mechanisms. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 32–43, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Aaron Block, Hennadiy Leontyev, Bjorn B. Brandenburg, and James H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Bullseye Testing Technology, http://www.bullseye.com/coverage.html. *Code Coverage Analysis*, 1996.

[4] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Litmus-rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126, Washington, DC, USA, 2006. IEEE Computer Society.

[5] R. Carver and K. C. Tai. Deterministic execution testing of concurrent ada programs. In *Proceedings of the conference on Tri-Ada '89: Ada technology in context: application, development, and deployment*, TRI-Ada '89, pages 528–544, New York, NY, USA, 1989. ACM.

[6] Jonathon Corbert. Cfs group scheduling. http://lwn.net/Articles/240474.

[7] Jonathon Corbet. Tty-based group scheduling. http://lwn.net/Articles/415740/.

[8] Yaniv Eytani. Concurrent java test generation as a search problem. In *Proc. of the Fifth Workshop on Runtime Verification (RV)*, volume 144(4) of *Electronic Notes in Theoretical Computer Science*, 2005.

[9] Dario Faggioli, Michael Trimarchi, and Fabio Checconi. An implementation of the earliest deadline first algorithm in linux. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1984–1989, New York, NY, USA, 2009. ACM.

[10] Geoff Gustafson, Julie Fleischer, Rusty Lynch, and Inaky Perez-Gonzalez. Open posix test suite. http://posixtest.sourceforge.net/.

[11] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[12] Wolfgang Karl, Markus Leberecht, and Martin Schulz. Optimizing data locality for sci-based pc-clusters with the smile monitoring approach. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 169–, Washington, DC, USA, 1999. IEEE Computer Society.

[13] Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 32(6):382–403, 2006.

[14] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[15] D. Niehaus. Hgs documentation. http://www.ittc.ku.edu/kurt/kusp_docs/gs_internals_manual/index.html.

[16] John David Regehr. *Using hierarchical scheduling to support soft real-time applications in general-purpose operating systems*. PhD thesis, Charlottesville, VA, USA, 2001. Adviser-Stankovic, John A.

[17] S. Rostedt and D.V. Hart. Internals of the rt patch. In *Proceedings of the Ottawa Linux Symposium*, pages 161–172, 2007.

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

[19] Scott D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

[20] Noah Watkins, Jared Straub, and Douglas Niehaus. A flexible scheduling framework supporting multiple programming models with arbitrary semantics in linux. In *Proceedings of the Real-Time Linux Workshop*, Dresden, Germany, September 2009.