Group Scheduling in Systems Software

Michael Frisbie and Douglas Niehaus
{frizz,niehaus}@eecs.ku.edu
EECS Department, ITTC
University of Kansas, Lawrence

Venkita Subramonian and Christopher Gill
{venkita,cdgill}@cse.wustl.edu
CSE Department, DOC Center
Washington University, St. Louis

Abstract

Previous system scheduling approaches have focused primarily on system-level abstractions for scheduling decision functions and the mechanisms used to implement them. This paper introduces a new abstraction called group scheduling that focuses primarily on the progress of application-level computations and on organizing system-level scheduling abstractions to ensure that progress.

This paper makes three contributions to system scheduling research. First, it defines a model for group scheduling that augments and complements hierarchical scheduling models. Second, it describes how a computation’s progress semantics can be mapped to scheduling mechanisms at the operating system and middleware levels. Third, it presents preliminary empirical studies of the performance of group scheduling in a realistic system environment.

1 Introduction

Computer system schedulers organize when and how computations sharing the system resources execute. Scheduling policy goals can vary with the nature of the system. General purpose systems emphasize computation progress and response time, batch systems often emphasize throughput, and real-time systems often emphasize timeliness and predictability.

Mechanisms used to achieve scheduling policy goals typically vary less widely than the policies themselves, i.e., relying on common services like POSIX thread priorities. The scheduling decision is thus reduced to selecting a highest priority thread from those ready to run. The idea is appealingly simple: integer priority values encode information the scheduling decision function uses to select the thread that should access the CPU at any given moment.

However, experience has shown that this simple model cannot describe the full range of execution constraints which computer systems need to enforce. One problem is that several threads may have equal priority values, so a secondary scheduling decision function (SDF) is needed to determine how the CPU should be shared among threads of equal priority.

Scheduling has been the focus of much theoretical and implementation effort over many years, producing a wide range of (1) different SDFs and (2) different ways to use them to implement an equally wide range of scheduling policies. The majority of these efforts have, however, been thread-centric in that they focused on SDFs that view the computational load on a system as a set of individual threads. Furthermore, most platforms available today have selected particular SDF semantics, without allowing system designers to influence how scheduling decisions affecting their applications are made, apart from specifying parameters used by the underlying system SDF.

This paper presents an approach to system SDF design and usage that addresses the limitations we believe are imposed by these typically thread-centric and monolithic system SDFs. We call our approach group scheduling because it explicitly addresses an important trend in system design: computations are implemented as related threads, and the most appropriate scheduling semantics for those threads depends on the semantics of the computation. Group scheduling addresses these issues by permitting threads to be grouped according to the computations they support, and by permitting each group to have an associated SDF. The system SDF is then formed by hierarchically composing group SDFs into a decision tree that controls the execution of all threads on the system.

This approach provides explicit support for a number of important design and implementation capabilities. First, it explicitly represents computational structure, so that the progress of a computation can be considered in scheduling decisions. Second, it increases
system customizability by permitting developers to select or implement the SDF that is most appropriate for each application computation and the system SDF that composes group SDFs. Third, it increases system transparency because logs from the system SDF offer a narrative of the scheduling decisions. Finally, group scheduling can be applied to concurrency abstractions other than operating system threads, e.g., events, distributable threads, or CORBA GIOP messages.

This paper is structured as follows. Section 2 describes related work. Section 3 presents the group scheduling model and gives examples of how it can be used. Section 4 describes our implementation of group scheduling in the KURT-Linux kernel and in middleware. Section 5 presents preliminary experiments. Last, Section 6 makes concluding remarks.

2 Related Work

Three main abstractions make up the group scheduling model presented in Section 3: the semantics for progress of each computation, the execution environment within which the computation is run, and the scheduling decision function used to schedule access to resources within each execution environment. Two areas of previous work are most relevant to group scheduling, and are the focus of this section: (1) making the execution environment flexible so that alternative SDFs can be expressed in it, and (2) composing SDFs for analytical assurance of real-time performance across multiple interacting execution environments. The main distinction between the work presented in this paper and the related work is that our work on group scheduling offers a general model for mapping computation semantics explicitly into the execution environments and SDFs, as a third dimension for design, measurement, and optimization.

A key theme in comparing our group scheduling abstraction to other related work is that of the dominant decomposition [14] that is used in each approach, i.e., the fundamental paradigm along which scheduling abstractions are defined. In Kokyu [3], the dominant decomposition was along the mechanisms for re-ordering the execution of schedulable entities. In the Scout operating system [6] and in TAO’s Real-Time CORBA 2.0 [8] (RTC2) implementation, the dominant decomposition is still along the execution environments, albeit those environments are customized for particular computation activities: paths in Scout, and distributable threads in RTC2. In the hierarchical scheduling frameworks [10, 4] and the BERT scheduling algorithm [2], the dominant decomposition is along the decision functions themselves. With group scheduling, the dominant decomposition is along the semantics of progress for the computation itself, as represented by the group structure. In group scheduling, SDFs are then composed along sets of paths constituting the execution environments within which parts of the computation are run. We view SDFs, execution environments, and scheduling mechanisms, as fundamental abstractions upon which our group scheduling abstraction builds. While functionally equivalent scheduling representations can be realized in each of these approaches in many cases, we view the dominant decomposition as a crucial issue for how these representations can be programmed in practice.

Execution Environments: The Kokyu scheduling framework [3] was designed to support flexible composition of middleware scheduling mechanisms for a range of SDFs. Kokyu maps the ordering policy of a given SDF into thread, queue, and task parameter configurations to enforce that policy.

The prototype implementation of TAO’s RTC2 implementation [5] introduced execution control mechanisms based on (1) thread priorities and (2) condition variables. Our group scheduling framework also includes condition variable and dynamic thread scheduling mechanisms, with detection of thread blocking and unblocking.

The Scout operating system [6] introduced the path abstraction, which made explicit the notion of the computation’s execution environment cross-cutting system layers. In Scout, paths compose segments of the computation with mechanisms that manage their progress. Our group scheduling approach generalizes the notion of path composition to a wider range of platforms, and also shifts the path abstraction from the execution environment model to the scheduling model: with flexible execution environments like KURT-Linux or Kokyu in which group scheduling mechanisms can be expressed, group scheduling raises the level of abstraction at which computation progress is modeled and enforced.

Scheduling Decision Functions: The work on flexible scheduling mechanisms in Kokyu was originally motivated by earlier work by Steward and Khosla [13] that introduced the maximum urgency first (MUF) SDF and showed that MUF could emulate several other well-known SDFs through different bindings for its parameters and those of the scheduled task set. Goyal, Guo, and Vin [4] and Regehr and Stankovic [10] extended the notion of a scheduling decision function to include composition of SDFs within a hierarchical scheduling framework. Regehr, et al., have extended hierarchical scheduling to cross-cut alter-
native execution environments so that analysis of both concurrency and schedulability can be done together [9]. The hierarchical scheduling model by Regehr and Stankovic also allows a directed acyclic graph (DAG) of SDFs - our current implementation of group scheduling assumes a tree structure, but we plan to extend our approach to SDF DAGs.

In the Scout operating system, other compositions of scheduling decision functions have been realized, such as the Best-Effort Real-Time (BERT) [2] algorithm. Both the hierarchical scheduling and BERT approaches achieve composition of SDFs across system layers. The BERT algorithm for slack stealing in fairness can be realized in the more general hierarchical scheduling model. Since group scheduling complements and augments the hierarchical scheduling model, it also extends the BERT approach.

3 Group Scheduling Model

The group scheduling model presented here has a simple structure, but addresses several important issues related to the semantics and structure of computer system scheduling. The components of the group scheduling model are: (1) scheduling decision functions, (2) group membership specifications, and (3) scheduling information for each member of a group. Furthermore, it is important to note that when more than one group is defined for the system, they are organized as a scheduling decision tree (SDT) for the system; the hierarchical organization of all schedulers in an end-system results in the system scheduling decision tree (SSDT). Each scheduling decision begins at the root of the SSDT and recursively descends through a series of SDF invocations until a decision is made. There are two ways to view the SSDT; as a first-refusal scheduling decision that calls the default system scheduler if no thread is identified, or as the sole system scheduling method. As Section 3.3 discusses, the latter view can be used to implement the former.

3.1 Scheduling Decision Functions

Each group has a (possibly unique) SDF associated with it. Any number of SDFs may be implemented, but most systems can be implemented using selections from a standard set. We have implemented a number of SDFs, including: static priority, dynamic priority, explicit plan, cyclic, processor share, round robin, EDF, and sequential. The interface for a scheduling decision function is quite simple: it takes the scheduling information describing the members of the group with which it is associated as input, and it returns a decision, which can take one of three forms: a thread ID, Pass or OS. The group and SDF API includes the ability to modify the scheduling parameters associated with each member of the group. For example, if the SDF of a particular group is priority based, then the group API gives access to the SDF API that makes it possible to change the priority of group members.

It is worth noting that the simple interface for the SDF has made it easy to provide implementations within which developers can select from a set of existing SDFs, or can choose to implement customized SDFs. This significantly adds to the flexibility and configurability of the systems using the group scheduling approach, compared to systems with a single default OS scheduler of any single type.

The SDF returns a thread ID when it has identified a specific thread as the one that should run next. If the current SDF is associated with the root group of the decision tree, then the system dispatcher switches context to the specified thread. If the SDF is associated with a group below the root of the scheduling decision tree, then the calling SDF recognizes the thread ID and returns it to its calling context until the root of the scheduling decision tree is reached.

The Pass return value identifies when the current SDF, and any SDFs below it in the SSDT were unable to identify a thread that should run. The most obvious reason for this to happen is that no threads managed by that portion of the SSDT were ready to run, but any given SDF might return this value for reasons of its own. When the called SDF returns Pass, the calling SDF is then free to choose another group member. If the SDF at the root of the SSDT returns Pass, then there is no thread managed by the SSDT that is ready to run.

The OS return value is a special token used to explicitly identify the choice of invoking the default scheduler of the underlying OS. This is useful when a set of processes controlled by a specialized SSDT should coexist gracefully with the normal set of computations on a workstation. For example, an explicit plan based SSDT might be used to control a set of real-time processes under group scheduling, but the schedule might include specific periods during which the set of computations controlled by the default OS scheduler should be given a chance to run. In this case, the explicit plan SDF would return a thread ID when the clock reached the planned execution time for each thread, and would return OS at the beginning of a period designated for the general OS operations.

If the SDF selects a member of the group that is itself a group, then the current SDF invokes the SDF of the selected group recursively. The recursive invocations of SDFs will terminate because the SSDT is a
Further development of the group scheduling may relax the single-parent-group requirement, permitting the hierarchy of the set of groups to be a DAG instead of a tree, but the recursive invocation of SDFs will be limited in either case.

### 3.2 Computations, Groups, and Decision Trees

The group scheduling model permits threads to join and leave groups at their own request, or at the request of threads running at an adequate authority level. Groups can be made members of higher level groups. This is how a SDT for a computation and the SSDT for the system as a whole can be constructed.

An interesting aspect of our group scheduling approach is that it makes it possible to separate the scheduling semantics for computation components from those controlling how system resources are shared among computations. Each computation is implemented using one or more threads, which can be organized onto one or more groups, forming the SDT for the computation. The higher level organization of the SSDT, above the level for the SDTs of the individual computations, can then be viewed as the SDT controlling how computations share the system.

This is not the only possible organizing principle for the SSDT, however, nor is it appropriate for all computations. For example, many computations have components with a variety of execution behavior constraints. Some components must meet strict timing requirements, while others can execute under less stringent requirements, or even on a best effort basis. In such a system, the SSDT might best be organized to group computation components with similar execution constraints together.

Not all interesting SSDT structures neatly partition the set of computation threads. While the simplest scheduling semantics limit each thread to membership in only one group, there are situations in which thread membership in multiple groups is helpful. One of the earliest motivations for group scheduling was the problem of how to ensure timely display of graphical data computed by a real-time process. We were able to ensure that the thread computing the change in the display ran according to its real-time constraints, while the display itself was updated much less predictably. After some measurement and thought we realized that this was because the display of the graphical data was under the control of the X server thread, whose scheduling was not being handled according to the real-time semantics. Making the X server thread a member of the real-time group has helped by ensuring that the X server is promptly given an opportunity to run. It is not a complete solution because as yet we have no way to ensure that the X server acts on the events generated by the real-time computation.

While we are still investigating the expressive limits of the the group scheduling model, it is already clear that scheduling decision trees can be used to describe a wide range of scheduling semantics for both individual computations and the set of computations in the system as a whole. We illustrate this expressive range with several examples in the next section.

### 3.3 Examples

This section illustrates some of the expressive power and range of the group scheduling model with two examples. The first is presented in Figure 1(a), and illustrates how the first-refusal group scheduling behavior mentioned earlier can be implemented. This SSDT uses the sequential SDF (Seq) as the root node to give (1) the group scheduling decision tree for the system a chance to choose a thread to run. If it fails to do so, returning either Pass or OS, then the (2) Linux SDF is called to select a thread. Figure 1(b) shows the SDT structure for a multi-level feedback priority queue. In the figure, the root of the SDT is a group using a priority based SDF, and whose members are groups containing threads that are in the same priority equivalence classes. Within each class, a round-robin SDF is used to evenly share the CPU among members of a class. The subtle aspect of this model is that the priority SDF must know about the priority of each thread, so that as the priority changes and a thread needs to shift from one equivalence class to another, the Priority SDF can cause it to do so.

### 4 Group Scheduling Implementation

The group scheduling model can be implemented either in the operating system or at the middleware level. The operating system level is attractive because it can support an implementation that is both
cleanly designed from the start, and efficient. Middleware implementations of group scheduling are attractive because they require less explicit cooperation from the operating system. Specifically, moving the group scheduling technique up to middleware makes it available for use with operating systems that do not already provide group scheduling services, and for which interested users do not have source access to the kernel. Realizing group scheduling mechanisms in middleware also leads to increased portability while still maintaining acceptable performance. In particular, middleware implementations of group scheduling can help to mask variations in the set of scheduling features available across operating systems, by using the available features to support middleware group scheduling mechanisms that provide more uniform behavior than the individual system's scheduling model. However, since the efficiency of middleware group scheduling mechanisms depends on (1) the scheduling mechanisms that are available from the OS, (2) the structure of the middleware itself, and (3) the nature of the scheduled entities, several implementation approaches are reasonable.

We currently have multiple implementations of the group scheduling model: one at the operating system level in KURT-Linux [12], discussed in Section 4.1, and several at the middleware level, though for brevity we only discuss one in depth in Section 4.2. The middleware approaches use the standard SCHED_FIFO scheduling class, thread priority scheduling, and condition variable mechanisms offered by Linux and other operating systems to manage the execution of threads under control of the group scheduling model. It is worth noting that in the implementations described in this section the majority of the code implementing the group scheduling model can be shared between the kernel and middleware implementations. The shared code uses the same data structures to represent group membership, support scheduling decision functions, track scheduling parameters for each member of each group, and implement the SSDT. The shared code uses the same data structures to represent group membership, support scheduling decision functions, track scheduling parameters for each member of each group, and implement the SSDT.

The only necessary differences between these kernel and middleware versions lie in how the group scheduling API is made accessible, how the system scheduling decision tree is invoked to identify the next thread to run, and how that decision is enforced. The two implementations we describe here therefore offer an opportunity to examine the fundamental costs of moving group scheduling from the kernel to the middleware layer. We describe results of our preliminary evaluation of these two implementations in Section 5.

After presenting the current kernel and middleware implementations, we consider briefly how to extend group scheduling to other schedulable entities at both the OS and middleware levels. It is important to note that not all entities will necessarily be visible at both levels. At the OS level, threads are often the only schedulable entities absent specialized OS support. A schedulable entity at the middleware layer, however, could be an OS thread, a published event in an Event Channel, a GIOP message, or a distributable thread.

### 4.1 KURT-Linux Implementation

The KURT-Linux implementation of the group scheduling model is quite simple, using the shared group scheduling code, except for specific adaptations to make the API available, invoke the SSDT, and enforce the scheduling decision. Group scheduling is implemented as a Linux kernel module, and provides access by threads running under the OS to the standard group scheduling API through a standard module interface. An application thread wishing to interact with the group scheduling API opens a file descriptor to the group scheduling module, and then executes various ioctl calls. Using this interface, groups can be created, threads made members of groups, and the group scheduling SSDT constructed.

The SSDT is invoked by small section of code inserted at the beginning of the Linux scheduler, which calls the SDF associated with the root node of the SSDT. The SSDT then returns a pointer to a thread structure, a Pass or OS value. If a specific thread is chosen by the SSDT, control is transferred directly to the context switch portion of the Linux scheduler, otherwise the standard Linux code selecting the next thread to run is executed. This approach is both simple and efficient. It requires a change to the Linux kernel code, but only of the smallest and least intrusive kind.

### 4.2 Middleware Implementation

The middleware implementation of the group scheduling model is more complex for two main reasons: first, because utility threads must be used to support the scheduling and group API functions; second, because indirect methods must be used to dispatch a thread, to control what threads can be run, and to arrange for the SSDT to be invoked. The group scheduling API is supported by a server thread listening for connection requests from threads wishing to interact with the group scheduling API. This is the middleware analog of the group scheduling module and the set of ioctl calls implementing the group scheduling API in the kernel implementation. Figure 2 shows the set of threads we use to implement
Figure 2. Middleware Group Thread Priorities

Group Scheduling Implementation Threads: The SSDT and API are concurrent threads in the same process. They run at the same priority, one less than maximum, because all changes to thread state that come through the API should be done before the scheduling decision is made by the SSDT. These operations could conceivably be implemented by a single thread, but we found using two more convenient. These are the operational threads that run at the highest priority, and thus are granted the CPU whenever they are ready to run. The API thread thus runs when a request for a group API operation is made, and the scheduler thread runs at times it selects according to the semantics of the SSDT it is running. In the experiments discussed in Section 5, we used a single level round-robin scheduler because our goal was to measure the overhead of thread management under the kernel and middleware implementations.

The controlled thread which is currently supposed to be running uses a priority two less than maximum, so it runs whenever the group API or SSDT threads are blocked, which is most of the time. The Block Catcher thread runs at three less than maximum so that it will be selected if the current thread becomes unable to run, which is most often when it is blocked. It can also run when the current thread exits. All other threads under the control of the group scheduling middleware are sent the SIGSTOP signal, and are given a priority of four less than maximum. This is somewhat redundant but we chose to do it for safety.

Choosing a thread to run is fairly simple. When the SSDT thread runs it invokes the system scheduling decision tree which decides what thread to run as described in Section 3. Assume for the purposes of this discussion that this is one of the threads under group scheduling control. The SSDT thread then decreases the priority of the current process to match that of other non-running threads under group scheduling control and sends it SIGSTOP. It increases the priority of the chosen thread to one less than maximum, and then blocks in a way that is determined by the SDF semantics. In the case of the experiments described here, it blocks in nanosleep because the SSDT thread is awaiting the end of the RR quantum.

If the current task uses the entire quantum, then the SSDT thread receives a signal for the nanosleep timer expiration and runs the SSDT to select the thread that should run. If the current thread blocks, then the Block Catcher thread is selected by the Linux FIFO scheduling policy and runs. The Block Catcher exists only to send a SIGUSR1 signal to the SSDT thread to cause it to return from the nanosleep early. The SSDT thread sends the now blocked thread SIGSTOP to ensure that it will stay stopped until selected by the SSDT.

Middleware Implementation Extensions: We have implemented two main extensions to the core middleware group scheduling implementation. The first of these addresses the problem of detecting when a blocked thread becomes unblocked. With open-source middleware frameworks like ACE [11], it is possible to intercept blocking system calls and alert the SSDT before and after each such call is made. This capability is particularly useful for correct scheduling of remote invocations in which scheduling parameters may be changed on the remote endsystem to which the call was made, and the changed values returned in the result, context, or output parameters of the call.

For applications that do not have an open layer at which blocking calls can be intercepted reliably, other alternatives are possible. In particular, we are investigating how to build mechanisms based on scheduler activations [1] to detect when a thread unblocks in such systems. For example, VxWorks [15] provides the capability for a thread to receive a signal when another thread blocks or begins running. This show promise as a basis for implementing a simple scheduler activation function, as long as the overhead imposed by this mechanism is reasonable. Therefore, we plan to follow implementation of those features with additional experiments similar to those described in Section 5 to assess that overhead.

The second main extension we have made is generalizing the set of schedulable entities to which group scheduling can be applied. In particular we are us-
ing the Kokyu framework to provide group scheduling capabilities for events, CORBA messages, and distributable threads. We are currently conducting studies to determine how the different semantics of these schedulable entities influence how group scheduling policies and mechanisms are applied.

5 Evaluation

The experimental results presented here give an initial evaluation of the thread management overhead under the kernel and middleware implementations. Several performance metrics are of interest, but this section only presents the one we consider most fundamental: the controlled thread to controlled thread switching delay, as illustrated in Figure 3 for the CPU bound tasks where the Block Catching thread does not run. This is defined as the delay between stopping a controlled task and starting another controlled task. As measured, it begins with the context switch away from the current controlled task (Context Switch A to SSDT) and ends with the context switch to the selected controlled task (Context Switch SSDT to B).

It is interesting to note that for the middleware implementation this delay could only be calculated by using data gathered from both the application and kernel levels via the Data Streams [7] method. The reason for this is that the context switch events come from the kernel, but the event announcing which thread was selected (Selection) by the SSDT comes from the SSDT thread at the user level. The post-processing of the event stream thus had to look for the event announcing which task had been selected in order to look for the proper context switch event.

In the kernel implementation measuring the time between one context switch and another would not be fair, because it would not include the SSDT execution time. So, in the a kernel implementation we measured the interval between when the SSDT is invoked and when the system switches context to the selected task. It is worth noting that the invocation of the SSDT follows the blocking of the first controlled task by only a few machine cycles. Figure 4 shows the distribution of delay values for the kernel group scheduling implementation controlling a set of blocking tasks. Note that the task to task switch delays vary between roughly 1.5 microseconds and slightly more than 2.6 microseconds. Also note that the vertical scale of all the histograms is exponential, so more than 1000 of the thread switch delays are classified in the first equivalence class of the histogram. The test was run on a 1.4 GHz Pentium machine, and shows a set of slightly less than 1800 context switch delay values. Clearly the simple RR SSDT evaluation imposes very little overhead, and the efficient context switch of the Linux platform is not compromised. It is important to note that the distribution for delay between non-blocking CPU bound tasks is essentially identical for the kernel implementation because the path taken through the kernel code is essentially identical.

In significant contrast to this, the middleware implementation controlling the same scenario with the same SSDT code experienced task switch delays, illustrated in Figure 5, ranging from slightly more than 110 microseconds to roughly 150 microseconds. It is interesting to note that there are significantly more larger values in this distribution. This is due, at least in part,
to interrupts occurring during essentially all of the longer task switch intervals, which we have been able to detect with fairly straightforward post-processing of the Data Stream output from the kernel.

![Chart](chart.png)

**Figure 6. Middleware Group Scheduling Task Switch Delay Distribution - CPU Bound**

Figure 6 shows the delay distribution for the middleware group scheduling implementation when controlling a set of CPU bound tasks. The distribution of thread switch delays covers roughly the same range as those for the set of blocking tasks; the lowest and highest values differ by roughly 40 microseconds. The range is, however, shifted lower by roughly 25 microseconds because with CPU bound threads context switching between tasks does not require the Block Catching task to run, a significant time savings.

6 Concluding Remarks

This paper has presented our group scheduling model and has described how it can be used in different situations. We have demonstrated that a middleware implementation is possible, with significant but not unsupportable task switching delay overhead. It has also shown that a kernel based implementation enjoys significant advantages over the middleware method, and is thus clearly desirable in systems where it is possible.

We plan to investigate how scheduler activations can be used to increase the efficiency of the implementation for blocking tasks. We also plan to show how additional semantics needed to complement common priority scheduling approaches, such as protocols to prevent unbounded priority inversions, can be integrated readily within a group scheduling SSDT. Moving from a notion of priority to more general SDFs allows the semantics of resource access and computation progress to be rationalized within an integrated group scheduling model.

References