User-Level Scheduling with Kernel Threads

Thomas Riechmann, Jürgen Kleinöder

June 1996 TR-I4-96-05

Technical Report

Computer Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University Erlangen-Nürnberg, Germany



User-Level Scheduling with Kernel Threads

Thomas Riechmann, Jürgen Kleinöder

University of Erlangen-Nürnberg
Department of Computer Science IV
Martensstr. 1, D-91058 Erlangen, Germany
http://www4.informatik.uni-erlangen.de/~riechmann
riechmann@informatik.uni-erlangen.de

Abstract

Today's operating systems provide kernel threads for parallel applications and multi-threaded servers. Scheduling plays an important role with regard to efficiency and fairness — especially for distributed applications, multimedia processing and server processes. A multi-threaded application should be able to specify the scheduling strategy for its threads itself. In most modern operating systems the scheduling strategy is hard-coded into the kernel and cannot be changed by the user. There are a few user-level thread packages available, where the user can define the scheduling strategy. Yet user-level threads are not suitable for applications that interact with the operating system frequently, such as server processes or distributed applications. In this paper we present a concept that allows handling of kernel-thread scheduling from the user level using hierarchical schedulers. Each application can have one or more of its own schedulers, which can define the application-specific scheduling strategy. Thus, the programmer can implement his own scheduling strategy for his application or even for subsystems inside the application. We have implemented our concept into Sun's object-oriented operating system Spring and we have shown that the overhead is extremely small.

1 Introduction

With the widespread use of distributed systems and multiprocessors, more requirements need to be met by operating systems these days. Modern hardware architectures are much more complex to manage than older monoprocessor systems. Additionally, the availability of more processing power leads to new and more complex applications — which in turn place even higher demands on the operating and runtime systems. This process creates a need for operating systems that are adaptable to a wide variety of hardware architectures and application demands. Examples of components that need to be flexible include security mechanisms and — especially in parallel architectures — memory and thread management.

Some newer operating system developments already deal with this problem. The Spring operating system [MGH+94], is an object-oriented operating system which provides support for convenient object interaction in a distributed system and multithreading within applications. Furthermore it comprises many user-replaceable server objects, thus it is adaptable to a wide range of requirements. Only very few services are hard-coded into the micro kernel and cannot be configured by the user. Unfortunately, one of these is the scheduling service.

As multithreaded applications become common, scheduling inside applications plays a very important role for efficiency and fairness. There are different categories of applications that especially profit from specific scheduling strategies:

(1) Calculating applications with small working set. These applications often consist of many threads. Threads rarely block. Scheduling can be important to make the working set fit into the cache.

(2) Calculating applications with large working set. These applications consist of many threads, which are often blocked because of page faults. Scheduling can be very helpful to minimize the working set and to prevent page thrashing.

(3) Distributed user applications that communicate across a network, for example through message passing, virtual shared memory, or object invocation. These applications consist of many threads, which are often blocked in kernel while waiting for a response from other application parts. Application-specific scheduling on one node can help to minimize the idle time of other nodes.

(4) Servers (daemon processes). Servers consist of many threads, which are often blocked (for example when waiting for the network). Scheduling is very important to guarantee fairness between different jobs and to gain maximum efficiency.

(5) Multimedia applications with different real-time demands. In many cases correct scheduling decisions are only possible if the quality-of-service parameters of the application are available.

All these applications consist of many threads and the scheduling strategy plays an important role. Optimal scheduling decisions depend on the current state and behavior of the application. For example, when running out of physical memory which is indicated by frequent thread blockings due to page faults, a scheduler could use larger time slices or suspend some of its threads for some time. Thus to implement an optimal scheduling strategy, a scheduler must be able to watch an application's behavior in the system and to easily interact with the application to keep track of its state and of the tasks the threads are going to fulfill.

In the next section we will try to match current concepts to these requirements.

1.1 Related Work

Most modern operating systems, like Spring [MGH+94], Solaris, Mach [Bla90] and Windows NT [Cus93] support kernel threads. These operating systems have support for highly multithreaded applications like server processes and calculating user applications. The scheduling strategy for these applications is hard-coded into the kernel and cannot be changed by the user.

Besides this in the last years a lot of user-level thread packages (e.g. Scheduler Activations [And92], FastThreads [And89] or QuickThreads [Kep93]) have been developed. In these packages user-level threads are built upon kernel threads, which are used as virtual processors. The problem with user-level threads is their incomplete integration into the operating system. As soon as a user-level thread blocks on a resource managed by the kernel or is preempted by the kernel, the application looses the control over its virtual processor and thus the control over scheduling. Special user-kernel communication and the instantiation of new kernel threads to maintain the application's number of virtual processors solve the most critical aspects of the problem (as shown with first-class user-level threads in Psyche [MLS+91] or scheduler activations [And92]). Nevertheless, the dynamic instantiation and deletion of kernel threads and the interaction between application and kernel imposes a lot of overhead if a large number of user-level threads are blocking frequently (sleeping threads [Kop95] are addressing this problem).

There are some newer, completely different operating system architectures which also allow user scheduling.

In SPIN [BSP+95] user-implemented schedulers can be downloaded into kernel. Thus the user can implement his scheduling strategy himself. While providing full flexibility, this concept suffers from the fact that the scheduler resides in the kernel. Intensive interaction with the corresponding application is rather expensive, because it needs cross-domain calls. Another problem is that the application-specific scheduler has to be written in Spin's system programming language (Modula-3) which may be different to the application programming language itself.

The Exokernel [EKT95] favors another concept. In the Exokernel the applications operate on the bare hardware as directly as possible. There are some problems with this approach, especially when applying it to the preemption mechanism of scheduling. If a thread is going to be preempted, a user-level routine is called. This routine has to save the registers. Because it is implemented by the application, a deadline is installed for it. The domain is killed if it does not return control before this deadline. Because the scheduling itself may take a longer time, this mechanism is not suitable for scheduling. In the exokernel implementation Aegis they give an example of how to implement a scheduling strategy at user level: They assign a thread that yields the processor and donates it to the thread that is to be activated. This costs an additional thread switch per switch and is especially inefficient when the scheduler activates the previous thread again.

1.2 Conclusion and Overview

The current approaches for user configurability of scheduling are not suitable for every kind of parallel application. Either the scheduling is not efficient enough or the schedulers are not able to communicate efficiently with the application. None of the concepts introduced is able to implement hierarchical scheduling, to allow different modules within an application to implement their own strategy.

In the next section we introduce a mechanism for obtaining complete user-level control over scheduling by moving the schedulers out of the kernel into user level. Although the design that we describe here is — in general — not tailored to a specific operating system and hardware architecture, our implementation for the Spring operating system [MGH+94] based on the SPARC processor led to several architecture-specific decisions. Spring's door mechanism depends heavily on thread-states in the kernel. Other-

wise it would have been possible to keep the thread-states on user level as in user-level thread-packages. As Spring is an object-oriented operating system, we shall describe schedulers, timers and other kernel components as objects.

2 Concept

As our concept for scheduling is rather general, we will present our ideas with very few Spring-specific issues. In the following sections we first introduce our general concept of hierarchical scheduling. Then we discuss some of the most important mechanisms in more detail, especially the switch mechanism itself, which includes timers, schedulers and preemption; we address security and we show how our concept applies to multiprocessors.

2.1 Scheduler hierarchy

Different user-implemented schedulers allow tailored scheduling strategies for application threads. These schedulers are arranged in a tree structure. For efficiency and security reasons it is advisable to keep at least the root inside the kernel. This system scheduler is normally used to switch between different applications and kernel subsystems. Each application that needs to implement its own scheduling strategy may instantiate its own scheduler in its address space. An application may instantiate more than one scheduler if it needs special scheduling decisions in some of its subsystems.

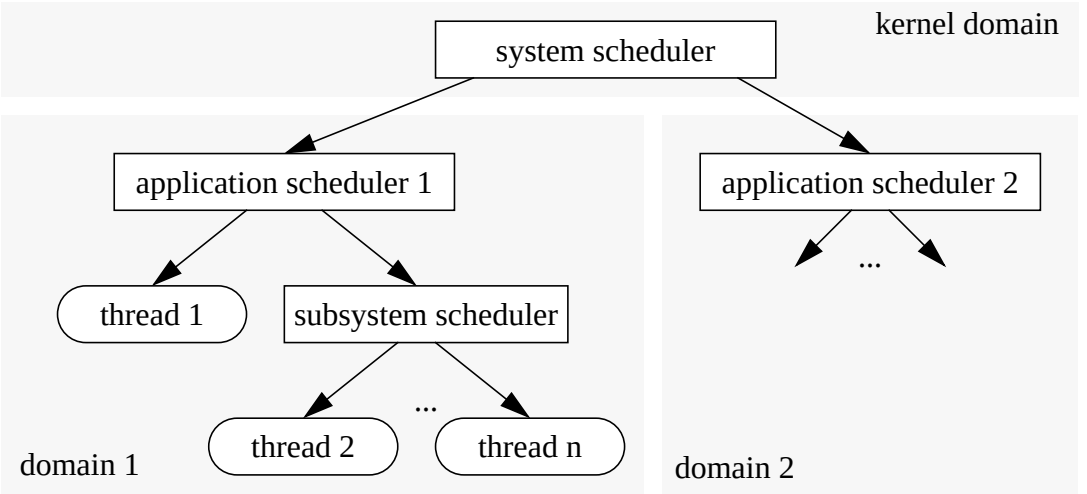


Fig. 2.1 Hierarchical schedulers

As shown in the example (Fig. 2.1) a scheduler can schedule threads and other schedulers. The application and subsystem schedulers for the application are located in user space. Thus the user can implement the scheduling strategy for his threads himself. As the Spring implementation needs kernel threads, one part of each thread object has to be kept in the kernel (in Spring it is named *shuttle*) and another part (e.g. the stack) is in user space (in Spring the *thread*). In the figures we place the thread objects into the domain where they are currently running.

2.2 Switching

Each scheduler manages a list of threads and/or schedulers. When it is activated, it decides which of them to run and which virtual time slice to pass to it. The scheduler gets the processor back after the specified virtual time slice. Each scheduler operates with the virtual time it gets from the scheduler above.

When a scheduler activates a thread or a scheduler, it marks this thread as *running* (or moves it from its *ready* to its *running* queue). As long as it is marked *running* the scheduler will not activate this thread or scheduler again (except on multiprocessors, see section 2.8).

2.3 Preemption

The central instance for activation and preemption is the timer: we have one timer object for each processor. Initially the timer owns the processor. It asks the system scheduler which thread or scheduler has to be activated (Fig 2.2, ①). The system scheduler returns a reference to the thread or scheduler it wants to activate and the time slice it should get. The interaction between timer and scheduler is done by method invocations (door invocation in case of cross-domain calls). The timer stores the time-slice information and activates the returned object: if it is a scheduler, it activates it by asking which thread or scheduler to activate (Fig 2.2, ②); if it is a thread, it activates it by switching to it (Fig 2.2, ③). The timer stores all time-slice information it gets from the schedulers before it finally gets a thread to switch to. So in our example the timer list finally looks like ④.

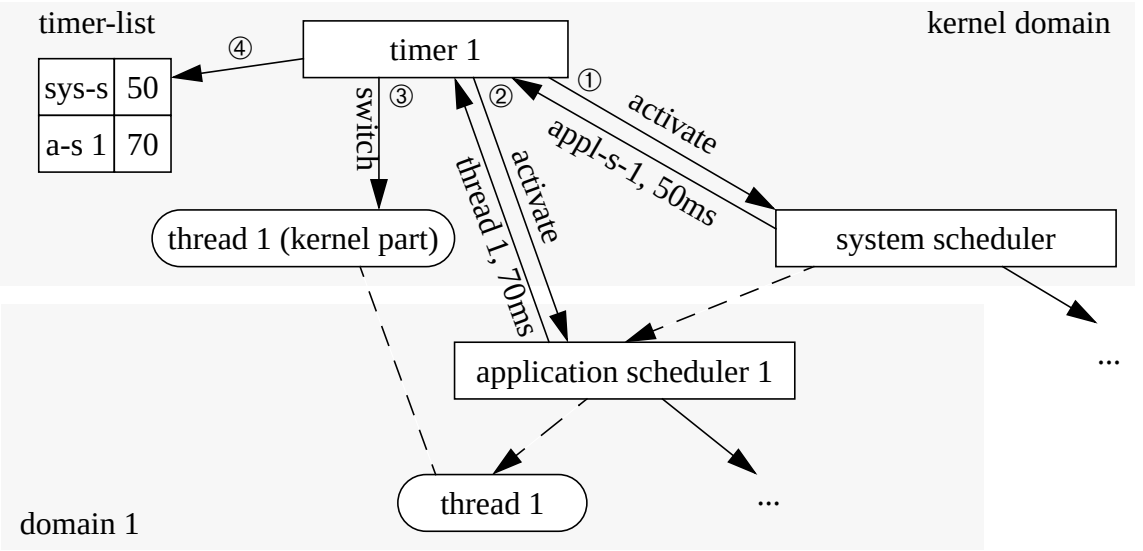


Fig. 2.2 Context switching

2.4 Hierarchical Timer

Now we will discuss the structure of the timer object in detail. The timer gets requests from each scheduler that wants to implement preemptive multitasking — as mentioned in the last section. The timer is able to store several hierarchical requests. Let us consider an example: In Fig. 2.2 we have two requests — one from application scheduler 1 (70 ms) and one from the system scheduler (50 ms). Thus the currently active thread (thread 1) is preempted after 50 ms. Its state and the lower state of the timer are saved

into its thread object and the system scheduler is activated and notified that the branch to application scheduler 1 is preempted. Thus the thread object (thread 1) represents the whole branch from the system scheduler to the application scheduler (Fig 2.3).

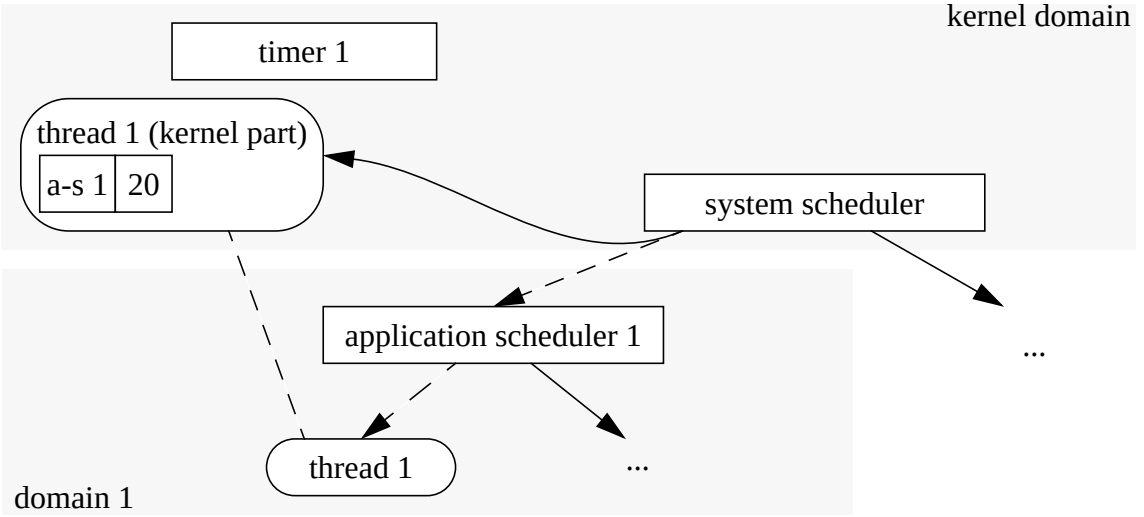


Fig. 2.3 The system scheduler preempts thread 1

If the system scheduler wants to reactivate the branch to application scheduler 1, it activates thread 1 directly by installing the saved timer list and the context of thread 1. The timer list is shown in Fig 2.4, ①. After 20 ms thread 1 is preempted by application scheduler 1. Thus the timer activates application scheduler 1. Of course the upper part of the timer list remains untouched (Fig 2.4, ②), so application scheduler 1 can only install an additional timer entry beyond the system-scheduler entry.

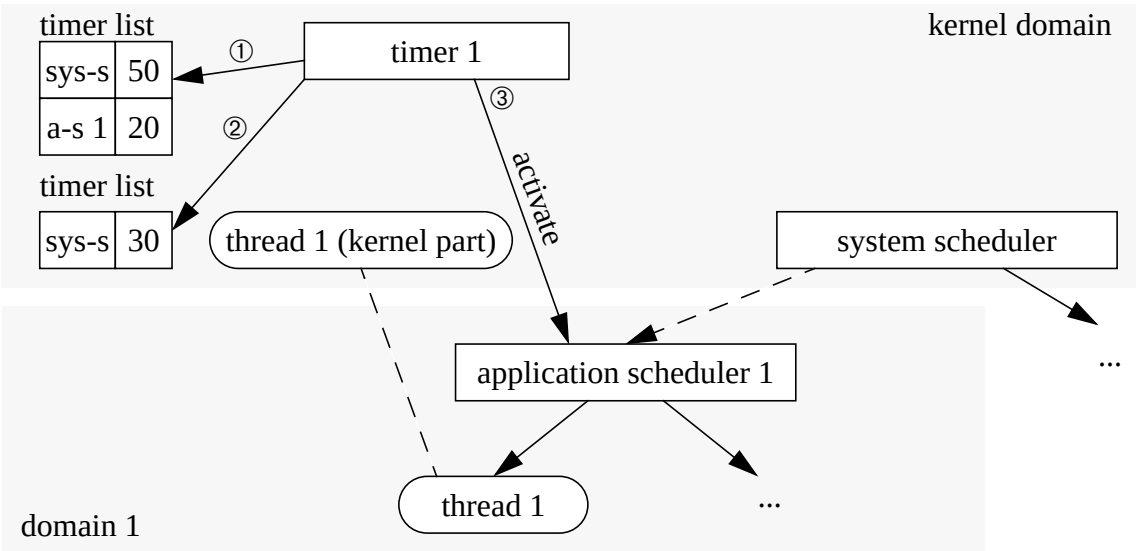


Fig. 2.4 Application scheduler 1 preempts thread 1

The resulting abstraction for the schedulers is:

- A scheduler contains references to branches which it can activate.
- If it activates a branch, it tells the timer how many time slices that branch should get. After the branch has used the time slices, the scheduler is notified and asked which branch should be activated next.
- It is transparent for the scheduler whether it or its child threads are preempted by a higher-level scheduler (e.g. the system scheduler). Each scheduler can choose its time slices individually.
- If a thread or a scheduler blocks or terminates, the same happens as in the case of preemption: the parent scheduler is notified and can decide which branch it wants to activate.

2.5 Security

There were several security problems that we had to solve. One problem is the call from the timer, when it asks a scheduler which thread to activate. It may be a call into user level, so the kernel is not sure that the user-level scheduler returns control; for example, the scheduler may be erroneous and may run in an endless loop. The solution to this problem is to make the call asynchronous. The timer generates a new thread ("switch thread") that asks the scheduler which branch should be activated; after return the selected branch is activated and the switch thread terminates.

While the switch thread runs, there may be an upper-level timer list installed (as in Fig 2.4, ②). If the switch thread is preempted or blocks it is installed into the branch of the preempted scheduler. For example in Fig 2.5 the switch thread which asks application scheduler 1 is preempted by the system scheduler. When the system scheduler activates the branch again, the switch thread is activated and no new thread is created.

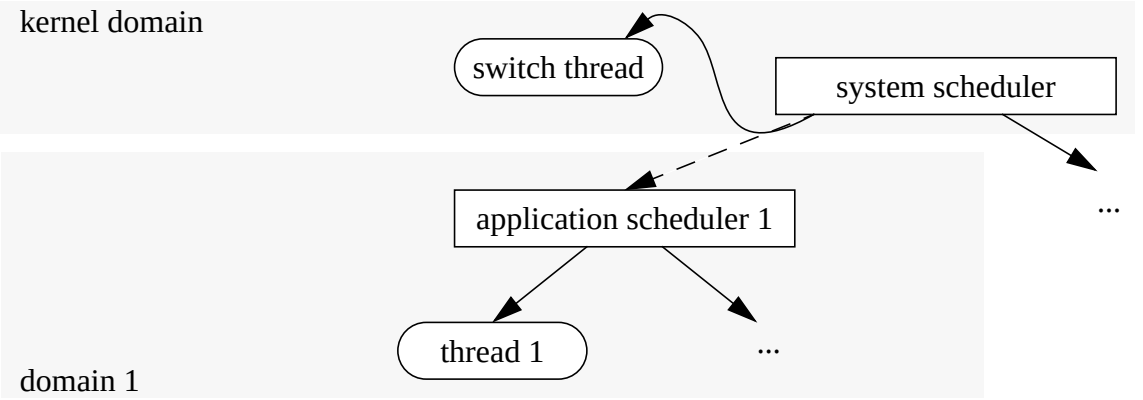


Fig. 2.5 Preempted switch thread

2.6 Cross domain calls

Another security aspect we have not mentioned yet is synchronous cross-domain calls. In Spring we have two different types of objects representing activity. We have “shuttles” which are the scheduling entities containing the saved processor state; and we have threads which contain the per-domain parts of the shuttle. If a thread calls into another domain, a new thread in the target domain is added to the single shuttle. Although in our model our schedulers only have references to thread objects, the shuttles are the scheduling entities. Thus normally the initiator scheduler in the calling domain schedules the call (Fig. 2.6).

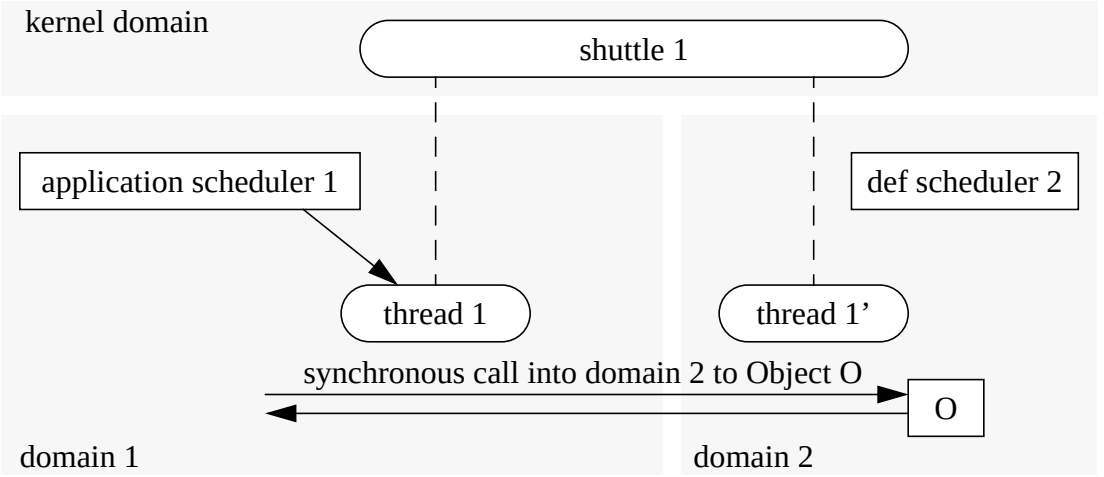


Fig. 2.6 Synchronous cross-domain calls

This is not acceptable for most applications and servers. If a scheduler could preempt a thread in another domain (in our example application scheduler 1 could preempt thread 1’), it could affect the functionality and integrity of that domain. If, in our example (Fig 2.6), thread 1’ allocates a lock in object O before it is preempted by application scheduler 1, the availability of the locked information depends on application scheduler 1, which could be user-implemented.

We solve this problem by passing the responsibility for the shuttle to the called domain. In our example application scheduler 1 would be notified that shuttle 1 left its domain and is no longer scheduled by it. A configured (perhaps user-implemented) default scheduler in domain 2 schedules it. When the call returns, the reference to shuttle 1 is again moved to application scheduler 1. (See section 2.7 for optimizations, which are essential here.)

This mechanism is especially important if a user calls into the kernel: A kernel invocation in Spring is a normal cross-domain call; we ensure with our mechanism that a thread that runs in kernel is always scheduled by the kernel.

2.7 Optimizations

Some aspects of our concept cause considerable overhead if they are implemented without further improvement. We optimized our implementation to gain efficiency. We will now discuss one of the most important optimizations of conceptual interest: the cross-domain calls.

A synchronous cross-domain call in Spring (door call) is implemented extremely efficiently, because it is the base mechanism for cross-domain interaction in Spring. Thus the overhead of four additional method calls (adding and removing the thread to/from the scheduler) is not acceptable. We solve this problem with a concept we call “lazy scheduling”:

We do nothing special as long as a thread runs. When the thread is preempted or blocks, we check if the thread is currently running in another domain (i.e. has changed its domain by a door call). If it is, we notify the two schedulers: the scheduler from the calling domain and the default scheduler from the called domain. Thus the scheduler of the calling domain conceptually schedules the thread until it is preempted by somebody (or it blocks), while it is running in the other domain. Only in that rare case is the additional scheduler interaction needed.

Because most cross-domain calls do not block and do not take much time, this will happen rather seldom.

2.8 Multiprocessors

As we have already mentioned, if a scheduler activates another scheduler, the activated branch is disabled in the upper scheduler. Thus the activated scheduler cannot get more than one processor. This is only suitable if we have a uniprocessor or if the scheduler only has one runable thread. On multiprocessors the scheduler can tell its parent scheduler, that it is able to handle n processors. Then the scheduler can activate it n times.

2.9 Conclusion

In our concept of hierarchical scheduling we tried to move as much as possible out of the kernel. The schedulers (except the system scheduler and the schedulers for the kernel subsystems) are located on user level and so communication between the application and the scheduler is cheap. Thus the scheduler can easily interact with the application and get information about the state of the application. This is important for optimal scheduling decisions.

We separated mechanism and policy: The switching mechanism is located in kernel, while the scheduling strategy is located on the user level.

We solved the security problems with very little overhead. Our concept of *lazy scheduling* helps to avoid overhead in most cases.

In contrast to other work our user-level schedulers always have complete control over their threads: User threads are never preempted in kernel. Even if a page fault occurs, the user-level scheduler is notified and the thread is marked as blocked in the user-level scheduler.

The only question is: How much overhead does complete user control of scheduling cost? This question will be answered in the next chapter.

3 Implementation

We implemented the concept described into Spring by completely removing the original scheduling mechanism from the kernel and replacing it with our own system scheduler. To maintain compatibility with existing Spring and Unix applications, we implemented a default (round robin) scheduler into the dynamically linked user-level Spring library. So the changes we made to the scheduling mechanism are transparent to existing applications — they run with our user schedulers as well (Fig 3.1).

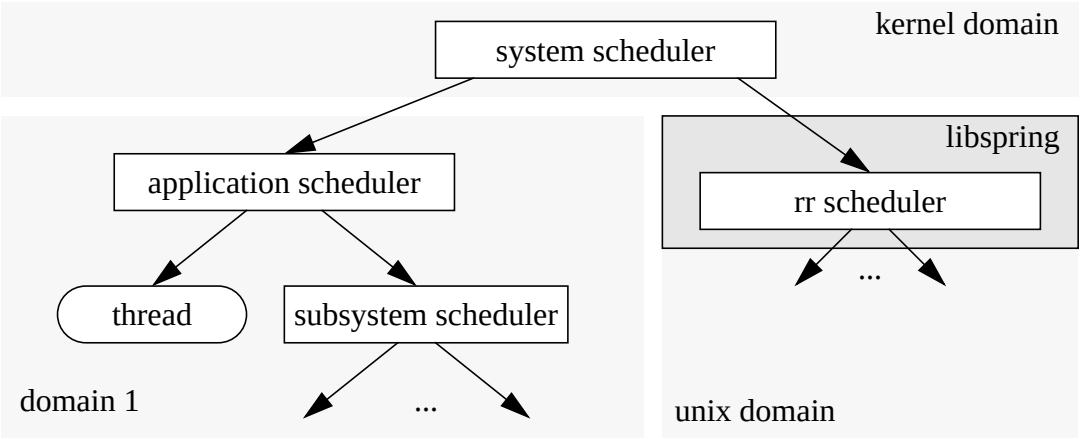


Fig. 3.1 Domains using our new scheduling concept vs. domains using defaults

In section 3.1 we will present some benchmarks of our schedulers. Then in section 3.2 we will present some configurations we tested, to show how our scheduling mechanism could be used in Spring.

3.1 Efficiency

Now we will present some benchmarks we obtained on a SPARCstation 20 with a Model 61 SuperSPARC SPARCmodule.

Action	Time (µs)
synchronous call into kernel	10
synchronous call into another domain	10
User-level scheduler activation (preemption or blocking of a thread)	11
ping pong with 2 threads and 3 mutexes (kernel scheduling)	300
ping pong with 2 threads and 3 mutexes (user scheduling)	250

Tab. 3.1 Benchmarks

Table 3.1 shows some benchmarks: The user-level scheduler activation costs (e.g. because of notification of blocking or preemption of one of the scheduler's branches) are nearly as cheap as for a cross-domain call. Thus the overhead costs of a user-level scheduler that uses 100ms time slices is only 0.01% of the computing time; with 10ms time slices it is only about 0.1%. Of course, if a scheduler decides to activate a thread, the thread switch itself needs some time, but that is the same time, the kernel scheduling needs.

We optimize if the scheduler decides to activate the same thread that was active before. In this case no real switch takes place, so there is no additional overhead. For the pure switch we do not have benchmarks, because the time needed varies too much: it depends on the number of register windows that have to be flushed. In Tab. 3.1 we see that switching is not very cheap (ping pong with 2 threads), because of the hardware architecture of the sparc processor (register windows).

Synchronization between two threads with user-level mutexes is about 20% more efficient with our user-level scheduling than with kernel scheduling, because the wakeup operation with user-level schedulers operates without cross-domain call, whereas the wakeup operation with kernel scheduling requires a trap.

3.2 Configurations

With our concept each application can implement the optimal scheduling strategies for its purposes. We now want to show two examples of how the concept of hierarchical schedulers can be used. Our first example is a calculating application (Fig 3.2). It contains a display thread, which is activated for example each second by the application scheduler, and several calculating threads, which are scheduled with large time slices (or perhaps non-preemptive) by a special calculation scheduler. The calculation scheduler could, for example, prefer threads that have made the least progress, to get maximum throughput on a multiprocessor. Because interaction between the scheduler and the state of the calculation is cheap (just a method call), there is not much overhead.

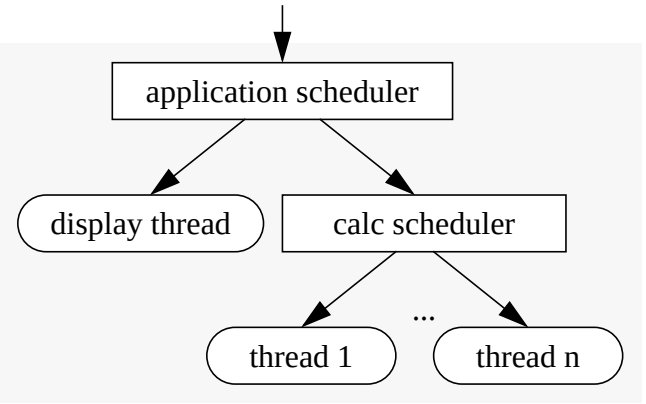


Fig. 3.2 Calculating application with hierarchical schedulers

This example has shown how one domain can benefit from our concept. But this concept is also useful for a user who wants his own scheduling between his different domains (Fig 3.3). The user instantiates one scheduler (for example in his login shell) and uses this scheduler to schedule all his applications (in our example a multithreaded x-server and a multithreaded text editor).

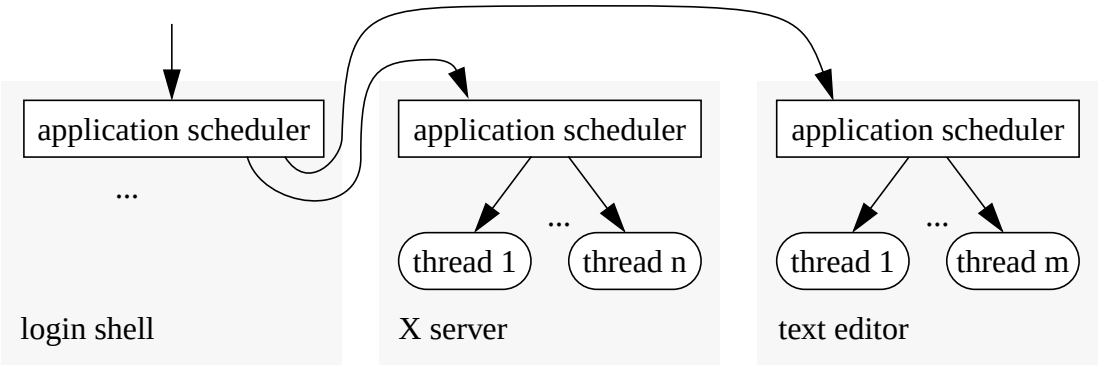


Fig. 3.3 Scheduling of applications from user space

4 Conclusion

User-level scheduling with kernel threads allows the user to implement the optimal scheduling strategy for his application, for his modules or even between his applications. Especially applications that operate with a lot of system interaction, such as distributed applications, servers and applications with large working sets need to define their own scheduling strategies to achieve higher efficiency. We can design class libraries with different kinds of schedulers and build special schedulers tailored to the needs of a specific application. Because scheduling is (in most cases) orthogonal to semantics, different schedulers can be tested with specific applications to obtain optimal performance. We tested our concept, for example, with a multithreaded calculating application (250 threads on a monoprocessor). We first tried the strategy “static priorities”. When we changed the strategy to round robin, we achieved about 15% higher performance. This application had a smaller working set with the round robin strategy, so the whole working set fit into cache.

We implemented our concept into Spring and we saw that the efficiency is in many cases better than with pure kernel scheduling — even without changing the scheduling strategy. When the scheduling strategy is adapted to the application, the resulting efficiency can be enormous.

References

And89 T. Anderson, E. Lazowska, H. Levy. “The Performance Implication of Thread Management Alternatives for Shared-Memory Multiprocessors”. *ACM Transactions on Computers*, Vol. 38, No. 12, pp. 1631 - 1644, Dec. 1989.

And92 T. Anderson, B. Bershad, E. Lazowska, H. Levy. “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53 - 79, Feb. 1992.

Bla90 D. L. Black. “Scheduling Support for Concurrency and Parallelism in the Mach Operating System”. *IEEE Computer*, Vol. 23, No. 5, pp. 35 - 43, May 1990.

BSP+95 B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, C. Chambers, S. Eggers. “Extensibility, Savety and Performance in the SPIN Operating System”. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, Dec. 1995, pp 267 - 284.

Cus93 H. Custer. *Inside windows NT*, Microsoft Press, Redmond, Washington, 1993.

EKT95 D. Engler, F. Kasshoek, J. O’Toole. “Exokernel: An Operating System Architecture for Application-Level Resource Management”, ”. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, Dec. 1995, pp 251 - 266.

HaK93 G. Hamilton and P. Kougiouris, “The Spring nucleus: A microkernel for objects”. In *Proceedings of the USENIX Summer 1993 Conference*, Cincinnati, Ohio, June 1993, pp. 147 - 159.

Kep93 D. Keppel. *Tools and Techniques for Building Fast Portable Threads Packages*. Technical Report UWCSE 93-05-06, University of Washington, May 1993.

Kop95 C. Koppe: “Sleeping Threads: A Kernel Mechanism for Support of Efficient User Level Threads”, *Proceedings of the Seventh IASTED ISMM International Conference Parallel and Distributed Computing and Systems*, Washington DC, Oct. 1995, pp. 11 - 20.

MGH+94 J. Mitchell, J. Gibbons, G. Hamilton, et.al.. “An Overview of the Spring System”. *Proc. of the Compcon Spring 1994*, San Francisco, California, 1994.

MLS+91 B. D. Marsh, M. L. Scott, T. J. LeBlanc, E. P. Markatos, “First-Class User-Level Threads”,In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 110 - 121, Pacific Grove, California, published as *ACM Operating Systems Review*, Vol. 25, No. 5, Oct. 1991.