Pragmatische nichtblockierende Synchronisation für Echtzeitsysteme

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Diplom-Informatiker Michael Hohmuth
geboren am 3. Februar 1973 in Annahütte

Betreuender Hochschullehrer: Prof. Dr. rer.-nat. Hermann Härtig
Technische Universität Dresden

Gutachter: Prof. Dr. rer.-nat. Hermann Härtig,
Technische Universität Dresden

Prof. Dr. sc. tech. (ETH) Gernot Heiser,
University of New South Wales, Sydney, Australien

Calton Pu, Ph. D., Professor and John P. Imlay, Jr. Chair in Software,
Georgia Institute of Technology, Atlanta, U. S. A.

verteidigt am 17. September 2002 in Dresden

Dresden, 2. Oktober 2002

Pragmatic nonblocking synchronization for real-time systems

Michael Hohmuth
Dresden University of Technology

October 22, 2002

Abstract

In this thesis I present a pragmatic methodology for designing nonblocking real-time systems. My methodology uses a combination of lock-free and wait-free synchronization techniques and clearly states which technique should be applied in which situation.

This thesis reports novel results in various respects: My approach restricts the usage of lock-free mechanisms to cases where the widely available atomic single-word compare-and-swap operation suffices. For more complex synchronized operations, I introduce a number of wait-free lock designs that work in different environments: uniprocessor and multiprocessor kernels, and user-mode programs. My multiprocessor lock implements a novel wait-free resource-access protocol—multiprocessor priority inheritance.

I show how monitors (such as Java's synchronized methods) can be implemented on top of my mechanisms, thereby demonstrating their versatility.

I describe in detail how I used the mechanisms for a full reimplementation of a popular microkernel interface (L4). My kernel—in contrast to the original implementation—bounds execution time of all operations. I report on a previous implementation of my mechanisms in which I used Massalin's and Pu's single-server approach, and on the resulting performance, which lead me to abandon this well-known scheme.

My microkernel implementation is in daily use with a user-level Linux server running a large variety of applications. Hence, my system can be considered as more than just an academic prototype. Still, and despite its implementation in C++, it is comparable in inter-process–communication speed with the original, highly optimized, non-real-time, assembly-language implementation, and it provides excellent real-time capabilities such as low interrupt-response latency.

Acknowledgments

First, I would like to thank my supervisor, Prof. Hermann Härtig. Despite scarce resources, he has grown the operating-systems group at TU Dresden to an internationally respected research institution and an enjoyable and inspiring place to work at. Without his continuing support and his wit, this thesis would not exist.

Many members of the operating-systems group at TU Dresden have contributed work that has helped making this thesis a reality. I am indebted to Jean Wolter, whose comments have opened my eyes more than once; Michael Peter, who helped improving Fiasco's synchronization primitives, worked on the multiprocessor version of Fiasco, and made Fiasco "really fast"; and Frank Mehnert and Sebastian Schönberg, who have helped me with performance measurements.

Jochen Liedtke, who passed away in June 2001, has been most influential to my interest in microkernel-based operating systems. I am grateful to him in many ways. He was an excellent teacher, mentor, colleague, and friend.

I am thankful to the many people who have read draft versions of this document or parts of it. Their valuable comments have helped improving the thesis tremendously. I thank Kevin Elphinstone, Claude-J. Hamann, Hermann Härtig, Gernot Heiser, Calton Pu, Thomas Roche, Sebastian Schönberg, and Jean Wolter.

Finally, I would like to thank those people that often are forgotten, simply because it is their job to make us feel them less: Adam Lackorzynski and Sven Rudolph, our current and former system administrators; and Angela Spehr, our group's secretary. Without them, I would not have had the time to write this thesis.

Contents

Chapter 1

Introduction

Nonblocking synchronization (with its two representatives, lock-free and wait-free synchronization) is a family of object-sharing protocols that prevent threads from resource-contention blocking. It provides two properties that are desirable for real-time systems: full preemptability and avoidance of priority inversion.

In recent years, nonblocking synchronization has caught the attention not only of the real-time systems community but of theoretical and some practical operating-systems groups. Many researchers have devised new methods for efficiently synchronizing a number of data structures in a nonblocking fashion. Others have conceived general methodologies for transforming any algorithm using blocking synchronization into a nonblocking one; however, these results have a more theoretical nature as the methodologies often lead to very inefficient implementations. The next chapter briefly discusses a number of these works.

In contrast to this boom, I know of only a few operating-system implementations that successfully exploit nonblocking synchronization. The only two operating systems I am aware of that use exclusively nonblocking synchronization are SYNTHESIS [MP91] and the CACHE kernel [GC96].

The following problems seem to thwart the adoption of nonblocking synchronization in practice: First, some related work induces the impression that special algorithms are needed to synchronize data structures in a nonblocking fashion, requiring abandonment of known programming paradigms such as mutual exclusion. Second, many of the most efficient algorithms available for lock-free data structures require a primitive for atomically updating two independent memory words (two-word compare-and-swap, CAS2), and many processors like the popular x86 CPUs do not provide such an instruction. Significantly, SYNTHESIS and the CACHE kernel originate from the Motorola 68K architecture, which does have a CAS2 primitive.

In this thesis, I present a pragmatic approach for building nonblocking real-time systems. This approach is based on two important insights: First, on uniprocessor systems (and for CPU-local data on multiprocessors) lock-freedom can be achieved not only through strong atomic primitives such as CAS2, but also by protecting critical sections from preemption—for example by disabling interrupts. Priority inversion does not occur as long as critical sections protected using this method are *very short*, that is, shorter than kernel operations that *need* to be atomic, such as context switches or kernel entries. Second, any implementation of priority inheritance provides wait-free synchronization as long as critical sections do not block.

Based on these insights, my methodology is very general and easy to apply. It is not limited to architectures that provide a CAS2 instruction, and it supports the mutual-exclusion programming paradigm.

My methodology consists of a set of rules for selecting nonblocking synchronization mechanisms based on usage patters of a program's objects. Low-overhead lock-free synchronization using very short critical sections or atomic instructions such as (single-word) compare-and-swap (CAS) is required only for accesses to global, performance-critical data. All other object accesses can be protected using locks with priority inheritance; wait-freedom is maintained by disallowing waiting for events inside critical sections.

By preventing sharing of resources using slow synchronization mechanisms between unrelated threads, my methodology supports both real-time and non-real-time applications at the same time on one system. Because these two types of applications are effectively decoupled with respect to the (kernel) objects they use, non-real-time applications cannot adversely affect the real-time properties guaranteed to real-time applications.

I present a number of lightweight wait-free lock designs for different system environments: uniprocessor kernels, multiprocessor kernels, and user-mode programs. Of these, the multiprocessor design implements a novel resource-access protocol: multiprocessor priority inheritance. I also discuss kernel interfaces that allow user-mode programs to take advantage of wait-free locking.

I describe the application of my approach to build a real system: Using my methodology, I developed the Fiasco microkernel, a kernel for the DROPS real-time operating system [HBB$^+$98] that runs on x86 CPUs. This kernel is an implementation of the L4 microkernel interface [Lie95], and it is sufficiently mature to support all the software developed for L4, including DROPS servers and L^{4}Linux [HHL$^+$97].[1] I evaluate the effectiveness of my methodology for nonblocking design by examining the Fiasco microkernel's real-time properties and synchronization overheads.

I also discuss a number of nonblocking synchronization mechanisms. In their SYNTHESIS work, Massalin and Pu [MP91] introduced the concept of a "single-server" thread (also known as the "serializer" pattern [HJT$^+$93]), which serializes complex object updates that cannot be implemented in a nonblocking fashion. In this thesis, I present a simple modification to the single-server scheme that makes it truly nonblocking and useful for use in real-time systems. Furthermore, I show that the single-server mechanism is semantically equivalent to a locking scheme. In particular, the real-time version can be replaced by a locking scheme with priority inheritance that is easier to implement and has a smaller performance overhead.

In this thesis, I do not address two related research areas that are outside the scope of this work. First, I do not digress into schedulability analysis for the wait-free locks I design in this thesis. While schedulability-analysis results for uniprocessor implementations of priority inheritance are widely known, to my knowledge such an analysis for multiprocessor priority inheritance does not exist yet. Second, I assume that multiprocessor real-time systems with mixed real-time and non-real-time loads are feasible. Clearly, a way of preventing potentially malicious non-real-time applications from using up too many of the shared resources in a multiprocessor systems, for example, of the memory-bus bandwidth, is required. Bounding such effects is a subject that is completely orthogonal to the topic of this thesis.

[1] L^{4}Linux is a port of the Linux kernel that runs as a user program on top of L4. Its application binary interface (ABI) is binary compatible with original Linux's.

I see my contribution in leading the recent interest in nonblocking synchronization to a practicable result, which the scientific community can verify. The source code to the Fiasco microkernel is freely available, allowing researchers to further study my techniques and experiment with them.

Organization of this thesis

This thesis is organized as follows.

In Chapter 2, I consider related work on microkernel-based systems, real-time operating systems, and nonblocking synchronization.

In Chapter 3, I develop my methodology for designing wait-free real-time systems (Section 3.1). I discuss implementation issues related to my methodology: In Section 3.2, I look at lock-free synchronization for kernels and user-mode programs. In Section 3.3, I propose a number of implementations for wait-free locking that work in different environments: uniprocessor and multiprocessor kernels, and user-mode programs. My multiprocessor lock (Section 3.3.2) implements multiprocessor priority inheritance, which I compare to multiprocessor priority ceiling. Also, I compare my synchronization primitives' strength to monitors (Section 3.3.4) and to a real-time–enhanced serializer (Section 3.3.5).

Chapter 4 shows how I applied my methodology to the development of the Fiasco microkernel. In Section 4.1, I specify functional and performance requirements that applied to the kernel, and derive design goals for the kernel. As the kernel is intended for use in real-time systems, the methodology developed in Chapter 3 can be applied. In Section 4.2, I showcase the use of my methodology in the context of Fiasco development.

In Chapter 5, I present performance values for the Fiasco microkernel. In Section 5.1, I compare the overhead of my in-kernel locking primitives with other forms of synchronization, including my real-time serializer. In Section 5.2, I evaluate the kernel's real-time properties. I compare its behavior under worst-case conditions with the behavior of RTLinux, a kernel that is known for its excellent real-time properties.

I conclude the thesis in Chapter 6 with a summary and suggestions for future work.

Chapter 2

Basics and related work

Related work to this thesis can be classified into the following three areas:

Synchronization in real-time systems. My thesis introduces a methodology for building real-time systems that combines existing work on resource-access control, including resource-access protocols (such as priority inheritance) and nonblocking synchronization. In Section 2.1, I discuss the problem of synchronization in real-time systems in general. I take a brief look at existing resource-access protocols and relate them to my work.

Nonblocking synchronization. Nonblocking synchronization methods have a number of properties that make them interesting for real-time systems. In Section 2.2, I review nonblocking synchronization mechanisms, reference existing operating systems that use this type of synchronization exclusively, and discuss the use of nonblocking synchronization in real-time systems.

Microkernel-based systems. This thesis is a systems thesis. Using my methodology, I designed and built a real system, the Fiasco microkernel, which is in daily use at Dresden University of Technology and elsewhere. In Section 2.3, I describe the state of the art in microkernel-based systems, and I look at existing work on real-time microkernels.

2.1 Synchronization in real-time operating systems

Classic real-time scheduling methods such as *earliest deadline first* (EDF) and *rate-monotonic scheduling* (RMS) [LL73] have ignored the issue of resources shared between multiple threads. These methods only allow sets of independent periodic processes that do not require inter-process synchronization. However, shared resources are clearly required in the design and implementation of nontrivial real-time applications.

In the past decade, researchers have proposed a number of solutions (many of which I cite later in this section) to allow resource sharing in both uniprocessor and multiprocessor real-time systems. These solutions either bound the time a process may block until it gets exclusive access to a shared resource, or they structure systems such that blocking cannot occur. In general, resource-sharing methods aim at preventing *priority inversion* where a low-priority thread blocking a high-priority thread is preempted by a mid-priority thread.

In this thesis, I only discuss methods that bound blocking by imposing protocols for resource access. I do not discuss methods that avoid blocking altogether. (I explain later in this section that, despite their name, "nonblocking synchronization methods" do not fall into the latter category.)

My thesis introduces an approach for building real-time systems that combines existing work on resource-access control, including resource-access protocols and nonblocking synchronization. I also provide a number of lock designs that—for different environments—implement one existing access-control protocol (uniprocessor priority inheritance) and one new protocol (multiprocessor priority inheritance).

In this section, I discuss some existing access-control protocols and relate them to the protocols I use in this thesis.

This section is organized as follows. Before I elaborate on synchronization protocols, I need to digress into the nature of blocking in Subsection 2.1.1. In Subsection 2.1.2, I discuss priority-based synchronization methods that bound blocking. Subsection 2.1.3 follows with a discussion of synchronization methods that are not based on priorities.

I discuss nonblocking synchronization separately in Section 2.2.

2.1.1 Blocking and blocking time

The term *blocking* refers to any delay that a released program[1] experiences because of resource contention (more than one program tries to access a resource at the same time) or resource unavailability. There can be various sources of blocking. For instance, the CPU on which the program is scheduled to run is used by another program (*release blocking*); the program requires exclusive access to a resource (other than the CPU) that has been locked by another program (*resource-contention blocking*[2]); or the program synchronizes on a signal from another program or an external device (*voluntary blocking*). The time during which a program cannot make progress because of blocking is called *blocking time.*

Real-time scheduling policies bound the total amount of blocking time a program can experience. Usually, such policies are designed to rule out certain kinds of blocking and enable system designers to reason about the remaining blocking factors (schedulability analysis). For example, priority-based scheduling policies (such as RMS) avoid release blocking of high-priority programs by independent lower-priority programs, and the priority-inheritance–based protocols (detailed in the next subsection) guarantee similar behavior even if the programs are not independent, that is, if the programs contend for exclusive access to shared (non-CPU) resources.

In this thesis, I consider only priority-based systems (the exception being my consideration of other related work in Section 2.1.3). As I outlined in the preceding paragraph, in such systems release blocking is easy to control. Therefore, I generally disregard release blocking. When I say that a program (or a thread) blocks, I imply that the blocking is either voluntary or caused by a synchronization protocol (resource-contention blocking).

BLOCKED PROGRAM VS. BLOCKED THREAD. Programs are executed by (or run inside) an execution thread (or just *thread*). A program is blocked when it makes no progress. A thread is blocked when it is descheduled from the CPU.

[1] A program is *released* if it has been started and is expected to make progress.

[2] In the literature, resource-contention blocking is also called *direct blocking*.

A blocked thread implies that the program running in the thread is blocked as well. However, I observe that a program can be blocked even if the thread in which the program runs is still executing on a CPU. If a thread experiences extra overhead to synchronize resource accesses, then the program executing in that thread is effectively blocked for the time of the overhead's execution. In particular, despite their name, nonblocking synchronization algorithms do not completely prevent blocking. These algorithms merely avoid blocking the current thread in case of a resource-access conflict. However, there is additional overhead associated with nonblocking synchronization (helping[3] cost or retry cost) that must be accounted for as resource-contention blocking time. In other words, the program is blocked even though the program's thread is not blocked. I go into more detail at the outset of Section 2.2.

2.1.2 Bounding blocking

UNIPROCESSOR PROTOCOLS. Of the methods that bound blocking time originating from shared resource access, the *priority-inheritance protocol* (PIP) and the *priority-ceiling protocol* (PCP) are most prominent [SRL90]. They are hard-priority schemes that fit uniprocessor systems with preemption and with potentially changing priorities. Both schemes are designed for periodic real-time systems in which threads consist of a periodic sequence of *jobs.*

PIP prevents priority inversion by temporarily lending low-priority threads the priority of high-priority threads that block on accessing a shared resource. PCP works by assigning resources priority ceilings that are equal to the highest priority of all users of the resource, and allowing resource access only if a thread's priority is higher than the ceilings of all currently locked resources. When an access is denied, the thread blocks and lends its priority to the thread locking the resource with the highest ceiling.

In both cases, priority inheritance—priority boosting for low-priority threads blocking high-priority threads—is used to block mid-priority threads in order to avoid priority inversion. This special form of release-blocking the mid-priority thread is called *push-through blocking.*

In comparison to PIP, PCP automatically prevents deadlocks and avoids extra worst-case execution time arising from "chained blocking"—nested critical sections in which a high-priority thread repeatedly needs to "help" low-priority threads by lending them its priority. Under PCP, a job can be blocked for at most the duration of one critical section.

However, PCP is more pessimistic in terms of resource access: It sometimes blocks high-priority threads when they access a resource even if that access would never lead to a resource conflict or a deadlock. This property can lead to blocking time that is higher than PIP's.[4] Another disadvantage of the PCP is that changing priorities is costly as the ceilings of the critical sections have to be recalculated and modified as well.

Many researchers have proposed refinements of PIP and PCP to work around these protocols' limitations. Audslay [Aud91] provides an excellent overview of these works.

[3]*Helping* is a family of mechanisms that implement wait-free synchronization. I explain wait-free synchronization and helping in detail in Section 2.2.1.

[4]In other words, there exists systems which are schedulable with PIP but not with PCP. The opposite is also true: As chained blocking cannot occur with PCP, some systems that are schedulable with PCP cannot be scheduled under PIP.

MULTIPROCESSOR PROTOCOLS. Rajkumar and associates [RSL88] proposed an extension of the PCP for multiprocessor systems, called the *multiprocessor priority-ceiling protocol* (MPCP). In that protocol, all resources are bound to fixed processors. The protocol distinguishes between local resources and global resources. Local resources can be accessed from only one CPU, whereas global resources can be accessed from any CPU. Threads have fixed priorities and are not allowed to lock more than one global resource (the latter limitation is not found in uniprocessor PCP). In order to access a global resource, threads notify a proxy server that runs on the resource's CPU (even if that CPU is the same as the thread's) and executes all critical sections using the resource. Proxy servers have a higher priority than all application threads. All threads running on one CPU (including proxy servers) and their accesses to local resources are scheduled according to the uniprocessor PCP.

Several authors have developed extensions to MPCP. Chen [Che95] and Rhee and Martin [RM95] proposed resource-access protocols that allow nested critical sections. Chen's protocols additionally allow dynamic thread priorities. Both works' authors show that their protocols lead to better schedulability than the original MPCP.

Unlike the resource-access protocols I have discussed so far, Jun Sun's end-to-end scheduling approach for multiprocessors [Sun97] uses a global view on the system. It allows tasks to have subtasks on multiple CPUs and computes schedulability for the system as a whole instead of separately for each CPU. End-to-end scheduling is not based on priority inheritance (it uses fixed priorities). However, it supports MPCP's synchronization model. Sun provided a mapping from MPCP systems' threads, priorities, and resources to end-to-end–scheduled systems. He developed a methodology for comparing the results of schedulability analysis for different resource-scheduling techniques. Using his methodology, Sun compared MPCP and his own approach and concluded that end-to-end scheduling provides better schedulability except for systems with a high number of short critical sections in each thread.

TECHNIQUES USED IN THIS THESIS. In later chapters, I use wait-free synchronization as one nonblocking synchronization mechanism. Because priority inheritance is an implementation of wait-free synchronization, I will consider only the priority-inheritance protocol (and an extension of it for multiprocessors). In Section 3.3 I will propose easy-to-implement lock designs with priority inheritance for a number of different environments: uniprocessor and multiprocessor kernels, and user-mode programs.

However, there is nothing in my methodology that precludes the use of other priority-inheritance–based resource-access protocols as long as they fit my methodology's design guidelines (defined in Section 3.1.2).

To my knowledge, my implementation of multiprocessor priority inheritance (MPIP; presented in Section 3.3.2) is the first such implementation. So far, approaches to the real-time multiprocessor resource-contention problem have focused on MPCP. Unlike MPCP and similar protocols, MPIP does not bind resources to fixed CPUs. Instead, it allows critical sections to be moved between processors. In consequence, MPIP works best for tightly-coupled multiprocessors. Another property of MPIP is that threads never need to wait for remote processors, guaranteeing steady progress independent of scheduling on remote processors.

In this thesis, I will not digress into the scheduling properties of MPIP or schedulability analysis. Such an analysis would be outside the scope of this thesis, where I use MPIP as one possible implementation of wait-free synchronization in multiprocessor systems. However, as MPIP is a wait-free resource-access protocol, schedulability analysis for wait-free synchronization schemes can be applied.

2.1.3 Other resource-sharing protocols

In my thesis I concentrate on resource-access protocols for systems using preemptive priority-based scheduling. In this section, I briefly look at two resource-access protocols that work without priorities, and I explain why I focus on priority-based systems. (I do not give specific references for the techniques I review in this section. Please refer to Audslay [Aud91] for a detailed overview.)

Reservation protocols avoid resource contention by having each task reserve the periodic interval in which it can lock a resource. The reservation protocol prevents reservations from overlapping according to some policy. For example, a policy can be based on fixed priorities: A task gets access to a resource only if it will release it before any task with higher priority requests that resource. However, it is also possible to use policies that do not use priorities, allowing this protocol to be applied to task systems without a clear priority order, for example interrupt-based systems.

Static scheduling uses a predetermined static schedule that completely avoids resource contention and blocking.

These resource-sharing protocols have in common that scheduling decisions are made in advance (pre-runtime), timing the execution of jobs such that resource-access conflicts are avoided. Priority-based systems, on the other hand, schedule in response to events. Event-based scheduling is more flexible than timing-based scheduling because it can change the schedule dynamically to react to changes in the execution environment. For example, when real-time application have not consumed their guaranteed worst-case share of CPU time, it is possible to increase resource utilization by running soft–real-time extensions or non-real-time applications (with a different set of resources), as we do in the DROPS system [HLR+01, HBB+98].

To summarize, priority-based scheduling lends itself to a wider range of applications, which is why I focus on such systems in this thesis.

2.2 Nonblocking synchronization

Nonblocking synchronization is a family of resource-sharing protocols that prevent threads from blocking because of resource-access conflicts. Nonblocking synchronization strategies have two important properties: First, they provide full preemptability and allow for multi-CPU parallelism. Second, priority inversion is avoided; lower-priority threads cannot block higher-priority threads because there is no blocking at all. These characteristics make nonblocking synchronization very interesting for real-time systems.

As I already remarked in Section 2.1.1, nonblocking synchronization protocols, despite their name, have synchronization overhead that needs to be accounted for as resource-contention blocking time. In other words, even though threads never block because of resource contention, the programs running in these threads do. Fortunately, synchronization overhead usually is very low, allowing nonblocking synchronization mechanisms to scale much better than blocking mechanisms [MS96]. Please refer to Section 5.1 to find detailed performance overheads for selected nonblocking synchronization primitives.

Nonblocking synchronization comes in two flavors: wait-free and lock-free synchronization. Both flavors have in common that they bound resource-contention blocking time at least for the highest-priority thread. Wait-free synchronization additionally bounds resource-contention blocking time for all other threads, whereas for lock-free synchronization this is true only under certain conditions (see Section 2.2.3).

This section is organized as follows. In Subsection 2.2.1, I describe the two flavors of nonblocking synchronization. I describe the nature and overhead of resource-contention blocking that is to be expected in these flavors, and I give an overview of techniques for atomic memory update, the availability of which is a precondition to implementing lock-free synchronization. In Subsection 2.2.2, I discuss existing operating systems that exclusively use nonblocking synchronization. Finally, in Subsection 2.2.3 I cite recent research that allows bounding resource-contention blocking time for all threads even for lock-free synchronization.

2.2.1 Wait-free and lock-free synchronization

2.2.1.1 Overview

Nonblocking synchronization comes in two flavors: wait-free and lock-free synchronization.

Wait-free synchronization can be thought of as locking, with *helping* replacing blocking. When a higher-priority thread *A*'s critical section detects an interference with a lower-priority thread *B*, *A* helps *B* to finish its critical section first, effectively lending its own CPU time to *B*. During helping, *A* also lends *B* its priority to ensure that no other, lower-prioritized activities can interfere. When *B* has finished, *A* executes its own critical section.

In wait-free synchronization, all shared-object updates are bounded in time (because helping ensures that each operation in a critical section is executed exactly once; there is no need for retries). However, all programs (including the highest-priority one) may experience some resource-contention blocking because they might need to help lower-priority programs to finish their critical section.

Wait-free synchronization mechanisms satisfy a stronger form of freedom from blocking than lock-free synchronization (discussed in the next paragraph) as they guarantee freedom from starvation. Therefore, some authors (e. g., Anderson et al. [ARJ97b]) point out that wait-free synchronization is a special case of lock-free synchronization. However, wait-free synchronization can also be implemented using locks, albeit with a nonblocking helping scheme. For example, a locking scheme with priority inheritance can be considered a wait-free synchronization scheme as long as critical sections never block.[5]

Lock-free synchronization works completely without locks. Critical code sections are designed such that they prepare their results out of line and then try to commit them to the pool of shared data using an atomic memory update instruction like compare-and-swap (CAS). The *compare* part of CAS is used to detect conflicts between two threads that simultaneously try to update the data; if it fails, the whole operation is restarted. To avoid retry contention on multiprocessor systems, retries can be delayed with a randomized exponential back-off. (Back-off is never needed on single-CPU systems.)

[5]When helping is implemented by priority inheritance, resource-contention blocking is replaced by release blocking. Depending on the implementation of priority inheritance, the system may temporarily switch to a different thread, but the total amount of blocking stays the same.

This synchronization mechanism has some nice properties: Because there are no locks, it avoids deadlocks (but not live-lock); it provides better insulation from crashed threads than locking schemes, resulting in higher robustness and fault tolerance, because operations do not hold locks on critical data; moreover, it is automatically multiprocessing-safe.

In lock-free synchronization on uniprocessors, the highest-priority thread always completes in one step and experiences no synchronization overhead. However, if it interferes with lower-priority threads (or with a thread running on another CPU), then these threads suffer from resource-contention blocking because they need to retry their operation in a *retry loop.* The number of retries is normally not limited; however, it is possible to construct systems in a way that such a limit exists—see Section 2.2.3.

Preconditions for using lock-free synchronization are that primitives for atomic memory modifications are available, and data is stored in type-stable memory. Type-stable memory is memory that cannot change type while a lock-free update is in progress. I do not digress into type-stable memory management in this thesis (see [GC96] for a discussion of operating-systems–related issues); the rest of this subsection discusses atomic memory modification.

2.2.1.2 Atomic memory update

CPU SUPPORT. Virtually all CPUs support a (multiprocessor-safe) primitive for atomic memory update. For example, x86 CPUs have three kinds of atomic memory-modification operations: a test-and-set instruction, a swap instruction, and a CAS instruction. Newer models (Intel Pentium and newer) also have a double-size–word (8 bytes) compare-and-swap instruction (CASW).[6] However, these CPUs do not support atomically updating two independent memory words (two-word compare-and-swap, CAS2).

CPUs with more modern instruction-set architectures, such as Alpha and MIPS CPUs, often do not provide atomic primitives that read and update a memory location in one step, because it is much more complex than a typical RISC operation and thus difficult to pipeline. Such CPUs provide an alternative mechanism based on two instructions, load-locked (LL) and store-conditional (SC). LL can be used to load a memory word into a register and "lock" the word's cache line. SC writes a word to memory only if its cache line is still locked. A cache-line lock is revoked when any CPU executes a normal load or store operation on that cache line.

LL and SC can be used to implement a slightly weaker version of CAS. This CAS operation can only succeed if its comparison succeeds *and* if no CPU accesses the affected cache line between the LL and SC instructions. This condition is slightly more general than that of a hardware-provided CAS instruction, for which a successful comparison is sufficient.

However, for all practical purposes, the CAS and LL–SC primitives are of similar power. Also, in practice one would restructure algorithms based on CAS to use LL and SC directly instead of a CAS emulation. Therefore, without any loss of generality, in my thesis I can concentrate on algorithms based on CAS; whenever I mention CAS, a construction based on LL and SC can be substituted.

A notable example of an architecture that does not provide CAS or a similar mechanism are SPARC CPUs older than the SPARC-V9; these CPUs only have an atomic (unconditional) swap instruction.

[6]The x86 assembly-language mnemonics for the CAS and CASW operations actually are `cmpxchg` and `cmpxchg8b`, but I will stick with the abbreviations CAS and CASW throughout this thesis.

SOFTWARE-IMPLEMENTED CAS AND CAS2. Bershad [Ber93] has proposed to implement CAS in software using an implementation and lock known to the operating system. When preempting a thread, the operating system consults the lock, and if it is set, it rolls back the thread and releases the lock. Greenwald and Cheriton [GC96] discuss a generalization of this technique to implement CAS2 or general multi-word compare-and-swap (MWCAS; a primitive that atomically exchanges multiple noncontiguous memory words, provided comparisons to “old” values succeed for each word). This method has the disadvantage of incurring overhead for maintaining the lock. Also, on multiprocessors, the lock must be set even when reading from shared data structures because otherwise readers can see intermediate states.

In Section 3.2 I will describe additional workarounds for providing (multi-word) atomic update in software.

SIMPLE DATA STRUCTURES. A number of data structures can be implemented and synchronized without locks directly on top of CAS and CASW (i. e., without the overhead of a software-implemented multi-word CAS): counters and bit fields with widths up to 8 bytes, stacks, and FIFO queues [Tre86, MS96].

Valois introduced a lock-free singly-linked list design supporting insertions and deletions anywhere in a list, as well as several other data structures [Val95a, Val95b]. These designs also work with just CAS. However, Greenwald [Gre99] has criticized them for being quite complex, difficult to get right, and computationally expensive.

Most of the algorithms for lock-free data-structure synchronization that have been developed recently assume availability of a stronger atomic primitive like CAS2. These data structures include general singly-linked and doubly-linked lists [Gre99].

A number of techniques exist for implementing lock-free and wait-free MWCAS on top of CAS and CAS2, enabling nonblocking synchronization for arbitrarily complex data structures [Her93, Moi97, ARJ97a, Gre99]. These techniques have considerable overhead in both space and runtime complexity, especially when compared to common lock-based operations, making them less interesting for kernel design.

MORE COMPLEX OBJECTS. For data structures more complex than those mentioned in preceding paragraphs, or when CAS or CAS2 are not available, atomic memory update must rely on other techniques.

Preventing preemption. The most common technique to implement atomic multi-word updates on uniprocessors is to prevent preemption during the update. This is usually done by disabling interrupt delivery in the CPU.

Where disabling interrupts is not possible (e. g., in user-mode programs), preemption can be prevented or delayed using operating-system support. This method works as follows: When entering a critical section, threads signal this condition using a flag shared with the operating system. Before the operating system preempts a thread’s execution, it checks the flag; if it is set, the preemption is delayed by a fixed time quantum (preemption is unconditional after the quantum has been used up). When a preemption has been delayed, the operating system sets another flag, signaling that a preemption is about to occur, and allows the thread to continue. The thread now can finish its critical section. Upon leaving the critical section, it checks the second flag and, if is set, voluntarily yields the CPU to prevent a forced preemption in its next critical section.

An extension of this mechanism is an operating-system feature that prevents descheduling only when it is in favor of a thread sharing data with the current thread. An operating-system–supported preemption-safe lock is an example of such a mechanism.

All of these methods work well when critical sections are known to be short. The maximum overhead of these methods is known and small, allowing them to be used in real-time systems.

The disadvantage of the first two methods is that they do not work on multiprocessors for global data. In this case, they must be extended, for example using a spin lock. Preemption-safe locks can support multiprocessors without extra measures.

Serializer approach. Another technique to facilitate complex object updates is the "serializer" or "single-server" approach [HJT$^+$93]. It uses a single server thread to serialize operations. Other threads enqueue messages into the server thread's work queue to request execution of operations on their behalf. If the server thread runs at a high priority, it does not block the requesting thread any more than if it had executed the operation directly.

2.2.2 Nonblocking synchronization in operating systems: SYNTHESIS and the CACHE kernel

Besides Fiasco, two other operating system projects have explored nonblocking synchronization in the kernel: the CACHE kernel [GC96] and SYNTHESIS [MP91].

Both systems run on architectures with a CAS2 primitive (the Motorola 68K CPU), and their authors found CAS2 to be sufficient to synchronize accesses to all of their kernel data structures. The authors report that lock-free implementation is a viable alternative for synchronization in operating-system kernels.

Massalin and Pu [MP91] originally also implemented a single-server mechanism for use in their lock-free SYNTHESIS kernel, but later they found no need to use it; the same was true for Greenwald and Cheriton [GC96] in their CACHE kernel. I will revisit the single-server approach in Section 3.3.5.

Greenwald and Cheriton [GC96] report that they found a powerful synergy between nonblocking synchronization and good structuring techniques for operating systems. They assert that nonblocking synchronization can reduce the complexity and improves the performance, reliability, and modularity of software especially when there is a lot of communication in the system.

However, they also warn that their results may not be applicable if the CPU does not support a CAS2 primitive. In this thesis, I will investigate how nonblocking systems can be implemented in such an environment.

2.2.3 Nonblocking synchronization vs. real-time systems

Nonblocking synchronization mechanisms are of interest for real-time systems because they provide preemptability and avoid priority inversion. It is well-known that wait-free method implementations are bounded in time (there is only a fixed number of threads and critical sections that potentially require help; no retry loop). However, it is not immediately apparent that this also applies to lock-free synchronization. On the surface, lock-free methods (like the ones in Figure 4.1 in Section 4.2.3) look dangerous because of their potentially unlimited number of retries.

Fortunately, Anderson and colleagues [ARJ97b] recently determined upper bounds for the number of retries that occur in priority-based systems. They derived scheduling conditions for hard–real-time, periodic tasks that share lock-free objects, and reported that lock-free shared objects often incur less overhead than object implementations based on wait-free or lock-based synchronization schemes.

2.3 Microkernel-based real-time systems

2.3.1 Microkernels: state of the art

2.3.1.1 The microkernel promise

As implied by their name, microkernels are small operating-system kernels. They came into being as the alternative to "fat," monolithic operating-system kernels that included all operating-system services such as scheduling, memory management, file systems, device drivers and more. In contrast to such monolithic designs, the microkernel idea is to allow replaceable user-level components, or *servers*, to provide operating system services. These servers run in separate address spaces and communicate using an inter-process communication (IPC) mechanism built into the kernel. This idea promised to enable a number of software-technological advantages: flexibility, modular system structure, uniform interface between components, better fault tolerance, smaller trusted computing base, multiple operating-system personalities [Lie96b].

Today, monolithic systems have evolved, and several of these advantages are not any more limited to the domain of microkernel-based systems. For example, many monolithic systems have a modular system structure, emulate multiple operating-system interfaces, and can be reconfigured flexibly at run time.

However, some of the advantages still hold and have specific applications:

- Microkernels isolate operating-system components from each other by running them in separate address spaces and providing them with separate resources. The main application of this feature is to run multiple applications with different requirements on one system:
  - It is possible to isolate real-time applications from time-sharing applications. For example, the DROPS system [HBB+98] (which is the main user of the Fiasco microkernel) runs real-time applications with resource guarantees side-by-side with L^{4}Linux, a complete time-sharing operating system [HHL+97].
  - Applications with security requirements can be isolated from notoriously buggy, easy-to-exploit time-sharing operating systems with potentially malicious applications. Therefore, it is possible to run security-sensitive applications on the same machine as the latest, Internet-downloaded games and viruses. Perseus [PRS+01] is one project that exploits this architecture.
- Microkernels are small. This property is of use in two contexts:
  - Microkernels can be used as a secure operating-system platform for custom, application-specific operating systems for small embedded devices.
  - The trusted computing base (TCB; the portion of system software that has to be trusted by security-sensitive applications) is potentially smaller than

with monolithic operating systems. This makes it easier to verify (or certify) security properties of the TCB.

In this thesis, I describe the Fiasco microkernel. Fiasco is mainly intended as a real-time kernel, but it has been used in all of the contexts I mentioned.

2.3.1.2 First-generation microkernels

First-generation microkernels unfortunately did not substantiate the hope for better-structured systems. Mach [GDFR90], the best-known first-generation microkernel, provides a good example of what went wrong with these kernels. Mach was designed to exploit modern hardware such as multiprocessors and enable new applications such as real-time applications while at the same time retaining compatibility with Unix. As a consequence, Mach was constructed by refactoring a Unix kernel, moving all Unix-specific functions to user-level servers and introducing new kernel interfaces that allowed both Unix and non-Unix operating-system servers to run as application programs.

While the Mach development resulted in numerous innovations such as external pagers, the result was a large kernel with hundreds of APIs that led critical researchers to ridicule Mach as the microkernel that is not micro. The major problem induced by Mach's size and functional richness, though, was that Mach's IPC was slow. As IPC is one of the most-often used services of a microkernel, Mach had a severe performance problem.

2.3.1.3 Second-generation microkernels

Learning from the problems of first-generation microkernels, designers came up with a set of design principles for new microkernels [Lie96b]:

- Only allow a minimal set of abstractions in the microkernel: address spaces, threads, and IPC.
- The microkernel should only provide mechanisms to multiplex the hardware securely. It should not implement itself policies for page replacement, message queueing, or device handling; it should allow these policies to be implemented at user level.

These principles led to a number of new microkernels that provide excellent performance both for system-level software and user applications. For example, in [HHL$^+$97] we showed that the L4 microkernel not only shows excellent microbenchmark performance, but incurs only minimal overhead in multiuser Unix application benchmarks—L^{4}Linux has an overhead of only 2 to 3 percent compared to monolithic Linux.[7]

Fiasco is an implementation of the L4 microkernel interface. In Section 4.1.1 I give an overview of the L4 interface.

[7] In [HHL$^+$97], we mention overheads of 5 to 10 percent. However, we later applied a number of further optimizations which reduced the overhead to 2 to 3 percent.

2.3.2 Microkernels in real-time systems

OTHER REAL-TIME MICROKERNELS. Real-time Mach [TNR90] is a real-time extension for the Mach microkernel. Real-time Mach includes real-time scheduling, synchronization, and IPC. It allows specifying policies governing the use of all resources managed by Mach, and offers a uniform resource model to describe the resource needs of applications to provide quality of service. In second-generation microkernels such as Fiasco, a resource description and reservation normally occurs in user-mode applications. For example, the DROPS system (which runs in user mode on top of Fiasco) includes a uniform resource model similar to Real-time Mach's [HBB+98]. Being a much higher-level, more monolithic kernel than Fiasco, Real-time Mach has been less concerned with low-latency event-handler activation, one of the real-time properties that my approach seeks to achieve.

There are a number of commercial real-time microkernels, including QNX [Hil92] and LynxOS [SB96]. The companies selling these microkernels publish interrupt latencies for their systems; however, little is known about the methods they used to achieve these results.

The Emeralds microkernel [ZPS99] is a real-time kernel intended for embedded applications. For IPC and synchronization performance, instead of relying on code optimization, Emerald's authors exploited the properties of embedded applications, such as small, memory-resident program code and up-front knowledge about task-communication patterns. Emeralds is unusual in that it does provide memory protection for user threads, but not virtual address spaces—a feature that is generally not required in embedded applications with a fixed task set. In contrast, L4 and Fiasco, are general-purpose microkernels.

CONSTRUCTION OF REAL-TIME MICROKERNELS. Elphinstone [Elp01] recently proposed a number of refinements of the L4-microkernel specification that enable the construction of an interesting, large class of real-time systems on top of L4. In particular, he specifies precedence and priority of a number of kernel services such as message-copying IPC to avoid priority inversion. Elphinstone does not prescribe a specific implementation for L4 interfaces, so his proposals are complementary to the content of my thesis.

The microkernel community recently discussed the "correct" way of implementing real-time microkernels.[8] Some community members speculated that complete kernel preemptability (as my approach provides, and as is implemented in Fiasco) is not required for short and predictable interrupt-response times. Microkernel operations are all short and "fast," so most kernel operations could run with disabled interrupts. This would not be a problem on multiprocessor machines, as most microkernel data structures are local to one CPU; where another CPU's data has to be manipulated, an inter-processor interrupt (IPI) would be used. For the longer operations that do occur in microkernels, strategic preemption points could be inserted in the kernel.

This proposal is justified; it is tempting because it avoids the need for a priority-inheritance mechanism. The proposal could be regarded as one extreme of the design space my approach opens up: As disabling interrupts is a valid lock-free synchronization technique, this approach uses only lock-free synchronization. In fact, the upcoming version of the L4 microkernel specification, dubbed "version 4," has been carefully engineered to include only very short critical sections. For example, in version 4, only

[8]Personal communication with Kevin Elphinstone, Volkmar Uhlig, and others at the Second Workshop on Microkernel-based Systems, Lake Louise, Banff, Canada, 2001

one task at a time can be deleted, whereas the specification implemented by Fiasco (version 2) requires that the task-delete operation also recursively deletes all tasks created by the task in question.

I believe that my approach is more general, because it is not limited to microkernels where most operations are very short (and, on multiprocessors, most operations use only CPU-local data structures). My methodology allows longer (preemptible) critical sections, freeing the programmer to structure complex algorithms such that safe preemption points can be introduced. Instead, existing code can be reused and wrapped into a critical section.

In practice, the cost of synchronization primitives, the frequency of cross-CPU communication, the expected level of data contention, and the length of critical sections determine which approach is to be preferred. In Section 5.1 I present performance overheads for a number of lock-free and wait-free synchronization primitives, and a more detailed discussion of this problem.

Chapter 3

Pragmatic nonblocking synchronization for real-time systems

In this chapter, I present a pragmatic approach for developing real-time systems using only nonblocking synchronization. My goal was to develop a methodology that leads to real-time systems with excellent real-time performance (i. e., low worst-case activation latency for real-time threads) and a low synchronization overhead.

The methodology I present is applicable to both kernels and user-mode programs on uniprocessors as well as multiprocessors. It is very easy to use; it uses a programming paradigm similar to mutual exclusion and uses only simple lock-free data structures.

This chapter is organized as follows.

In Section 3.1, I define the problem that my approach is intended to solve, and I develop the guidelines that comprise my methodology. From these guidelines, I derive further requirements, which I will discuss in the remaining sections of this chapter.

Section 3.2 discusses lock-free synchronization (and priority-inversion–free atomic update in general), one of the two nonblocking methods I recommend in my methodology. I look at ways to provide the required atomic primitives for both kernels and user-mode programs.

Section 3.3 discusses wait-free synchronization, the other nonblocking method I recommend in my methodology. In this section, I propose using locks with priority inheritance as the wait-free synchronization primitive. I present a number of different lock designs for use in different environments: uniprocessor and multiprocessor kernels, and multithreaded user-mode programs. I explain multiprocessor priority inheritance, the resource-access protocol my multiprocessor lock implements, and compare it to multiprocessor priority ceiling. Also, I compare my synchronization primitives' strength to monitors and to a real-time–enhanced serializer.

3.1 A design methodology for real-time systems

3.1.1 Design goals

My main design goal was to allow operating systems to have good real-time properties. Specifically:

- The operating system needs to provide static, “hard” thread priorities. Priority inversion must be avoided.
- Higher-priority threads must be able to preempt lower-priority threads (independent of whether they execute in user mode or kernel mode) at virtually any time, as soon as they are ready to run—thus allowing for good schedulability of event handlers. This should be true for sets of threads that depend on common resources, but even more so for threads that do not.

Secondary goals are:

- Short critical sections working on global state must induce essentially no overhead for synchronization.
- The design must be applicable to modern CPUs, that is, it must be implementable without CAS2.
- The synchronization schemes should work for both uniprocessor and multiprocessor architectures.
- The design must allow real-time and non-real-time tasks on the same CPU, without impeding the real-time properties guaranteed for real-time applications.

I have derived these goals from the specification of the Fiasco microkernel (see Section 4.1 and [Hoh98]). Many hard–real-time systems have similar requirements.

3.1.2 Design guidelines

The primary design goals rule out any synchronization scheme that suffers from priority inversion. This requirement led me to look into nonblocking synchronization schemes: lock-free and wait-free synchronization.

The secondary goals strongly favor lock-free synchronization schemes: Locks induce overhead, and in the multi-CPU case, the CPUs would compete for the locks. I therefore generally disallow lock-based schemes for frequently-used global state, except where completely lock-free synchronization is not possible on multiprocessor systems.

In particular, my design methodology comprises the following guidelines:

1. **Arrange a system’s threads** into groups of threads that cooperate on a given job or assignment. I call threads that belong to the same group *related,* whereas threads belonging to disjoint groups are *unrelated*.
2. **Classify a system’s objects** as follows: *Local state* consists of objects used only by related threads. *Global state* consists of the objects shared by unrelated threads.

3. **Frequently-accessed global state** must be implemented with data structures that can easily be accessed using *lock-free synchronization* or another low-overhead priority-inversion–free mechanism for atomic update.

   In Section 2.2.1, I mentioned a number of data structures that can be synchronized in this fashion using only CAS: Counters, bit fields, stacks, and FIFO queues.

   On uniprocessors (and for CPU-local data on multiprocessors), preventing preemption (e. g., by disabling and enabling interrupts in the CPU) for very short critical sections is a valid lock-free technique. *Very short* here means that critical sections do not use more CPU cycles than the most expensive atomic CPU instruction.

   On multiprocessors, with many CPUs lacking any lock-free synchronization mechanism better than single-word CAS, I discourage the use of global data structures that cannot be synchronized using CAS. However, when such data cannot be avoided, I propose it is also implemented in a lock-free fashion, based on a software implementation of MWCAS, or synchronized using very short, lock-based critical sections.

4. **Global state not relevant for real-time computing, and local data** can be accessed using *wait-free synchronization.* This kind of synchronization has some overhead. Therefore, it should be avoided for objects that otherwise unrelated threads must access.

   I propose using a locking mechanism that provides priority inheritance. Waiting for events (i. e., voluntary blocking) inside critical sections is not allowed. This restriction ensures wait-freedom.

Once a designer has decided which object should be synchronized with which scheme, this methodology becomes very straightforward to use. It approximates the ease of use of programming with mutual exclusion using monitors while still providing the desired real-time properties.

The set of rules I just determined ensures that the design goals of Section 3.1.1 are addressed:

No priority inversion; preemptability. By using hard priorities and only nonblocking synchronization (and very short critical sections), the system is preemptible, and priority inversion is avoided.

Low overhead; real-time and non-real-time applications. By disallowing expensive, lock-based synchronization schemes for data shared by all threads (i. e., by both real-time and non-real-time threads), manipulating global state induces only low overhead.

Independence from CAS2. Allowing preemption prevention as a lock-free technique allows the methodology to be used for uniprocessor systems with CPUs that do not provide CAS2, such as the x86.

On multiprocessors, where preventing preemption does not suffice, global data that cannot be accessed using only CAS must be synchronized using another low-overhead technique, such as a spin lock or a software implementation of MWCAS.

Uniprocessors and multiprocessors. All synchronization primitives can be provided for both uniprocessors and multiprocessors (see next two sections).

User mode and kernel mode. The techniques work for both kernels and for user-level programs.

The methodology imposes the following further requirements on systems, for which I will propose solution in subsequent sections:

- Efficient mechanisms for priority-inversion–free atomic update of global state: preventing preemption (for data accessed on only one CPU), and a software implementation of MWCAS or very short lock-based critical sections (for data accessed on multiple CPUs).

  I discuss atomic update for kernel mode and user mode in Sections 3.2.

- An implementation of priority inheritance in the kernel. I propose a wait-free priority-inheritance locking mechanism that can be characterized as "locking with helping," explained in more detail in Section 3.3.1.

  In Section 3.3.2, I extend my locking mechanism to multiprocessors. This lock design implements a novel resource-access protocol, multiprocessor priority inheritance (MPIP), which I also explain in this section.

- The priority-inversion–free locking mechanism must be available from user space. Section 3.3.3 discusses possible interfaces.

- Wait-free synchronization requires that critical sections protected by priority-inversion–safe locks must not block.

  I will show in Section 3.3.4 that this restriction does not limit the synchronization mechanism's power or ease of use.

  For user-mode programs, guaranteeing absence of blocking can be challenging. I discuss this problem in Section 3.3.3.4.

Using these guidelines, I have developed a real system—the Fiasco microkernel. Chapter 4 discusses Fiasco as an extensive example of the application of my methodology.

3.2 Priority-inversion–free atomic update

Previous section's methodology required an efficient implementation of low-overhead priority-inversion–free atomic update for global data. It strongly favored lock-free synchronization, but called for alternative mechanisms where lock-free synchronization is not applicable. In this section, I review atomic-update mechanisms that can be applied with my methodology.

As I detailed in Section 2.2.1.2, most CPUs offer atomic CAS instructions (or LL–SC instructions, which are of similar power).[1] (On x86 CPUs, these instructions also work across CPU boundaries in multiprocessors, i. e., maintain cache and memory consistency, when preceded by a `lock` prefix.) Using CAS, it is possible to implement

[1]On CPUs that do not provide CAS or LL–SC (or a primitive of similar power), one of the workarounds I provide in the following subsections can be used even for single-word atomic update.

in a lock-free fashion many simple data structures such as counters, FIFO queues, and stacks.

However, more complex data structures, such as doubly-linked lists, need stronger atomic-update primitives that allow changing multiple memory words at once. I discuss multi-word atomic update in the following two subsections. In Section 3.2.1, I consider kernel-mode atomic-update implementations for both uniprocessors and multiprocessors. I will consider user-mode implementations in Section 3.2.2.

3.2.1 Atomic update in kernel mode

On a uniprocessor, multi-word atomic update (such as very short critical sections, or a software implementation of MWCAS) can be protected by disabling interrupts in the CPU as I mentioned in Section 2.2.1.2. This technique has very low overhead because of two reasons: First, it does not need significantly more CPU cycles than what an atomic instruction updating the same number of memory words would require. Second, the technique does not need main memory for synchronization, reducing cache load.

Note that disabling interrupts in the CPU also works on multiprocessors for CPU-local data. However, it does not help for global data shared between multiple CPUs because it only affects the current CPU, not all CPUs, and because all other CPUs can still read the modified data and see intermediate states.

On some hardware architectures, disabling interrupts in an external interrupt controller is an alternative for synchronizing accesses to global data. For example, x86 multiprocessor systems can disable external interrupt sources using the IO-APIC chip or the old-fashioned PIC chip; in both cases, it is not possible to globally disable interrupts generated within CPUs, such as timer interrupts generated by a CPU's Local APIC unit. Unfortunately, this method has much higher overhead than a spin lock (both in the contented and uncontented case), making this synchronization scheme unpractical on this architecture.

Other possible options for multiprocessors are (all but the first recalled from Section 2.2.1.2):

1. Use spin locks to protect very short critical sections; these critical sections can contain an MWCAS implementation or object-specific code. Additionally, preemption of threads holding locks must be avoided. In a kernel, the simplest way to prevent descheduling is to additionally disable interrupts in the local CPU during critical sections.

2. Use Greenwald's and Cheriton's globally-locked software CAS2 or MWCAS with roll-back [GC96].

3. Use Anderson's and colleagues' wait-free MWCAS implementation [ARJ97a].

The first two methods use a lock to protect critical regions. In both cases, read operations as well as write operations have to be protected using a lock to prevent reads from seeing intermediate states. For both methods it is possible to reduce the number of data objects per lock, thereby reducing lock contention and improve scaling [Gre99].

Solution 3 is more complex and suffers from contention for a shared version variable. It has the advantage over Solution 2 that it does not need rollback in case of preemption. However, my methodology mandates lock-free synchronization—and thereby, the potential for rollback—on the level of MWCAS users, which makes wait-freedom on the MWCAS-primitive level a moot point.

I have not looked into solutions that are even more complex or have higher overhead than wait-free MWCAS.

Solution 1 (spin locks, critical sections with disabled interrupts) is the easiest to use and implement: It has the lowest potential for contention (because there can be as many spin locks as protected objects), it is not complex, and it is well-understood. All in all, I have not found a compelling reason to prefer a software-emulated MWCAS operation over using spin locks for very short critical sections. Therefore, I propose that it be used on multiprocessors.

3.2.2 Atomic update in user-mode programs

Multi-word atomic update can be emulated in software only if it is possible to disable concurrent access to the shared data. The interrupt-disabling method to prevent preemptions does not work at user level. Therefore, disabling concurrent access implies some kind of locking. However, the selected locking mechanism must not lead to priority inversion.

In general, the locking mechanism depends on the underlying operating system. As critical sections accessing data that is updated using lock-free synchronization are typically very short, it is important that a locking mechanism induces very little overhead—at best, the kernel should not need to help. I know of the following locks that fit this constraint:

1. Spin locks can be used on multiprocessor systems that always gang-schedule all of the program's threads.

2. The operating-system–assisted MWCAS implementation and the delayed-preemption mechanism (both of which I discussed in Section 2.2.1.2) are applicable on uniprocessors.

3. Operating-system–assisted preemption-safe locks are locks that prevent uniprocessorea lock-holding thread in favor of a thread that shares the lock-protected data structure. These work well on both uniprocessors and multiprocessors.

3.3 Wait-free synchronization

In Section 3.1, my methodology recommended using wait-free synchronization for local data and global state not relevant for real-time computing. In this section, I look at ways of implementing wait-free synchronization in different environments: uniprocessor and multiprocessor kernels, and user-mode programs.

In related work (see Section 2.2.1), wait-free synchronization often is implemented using complex and difficult-to-use MWCAS primitives. However, many authors overlook that mutual exclusion with priority inheritance is a valid wait-free synchronization scheme as long as critical sections never block. As mutual-exclusion–based interfaces are considerably easier to use, I concentrate on locking with priority inheritance in this thesis.

In this section, I propose a number of wait-free synchronization primitives for different environments. These primitives implement priority inheritance using a mechanism called *helping.* I call the primitive mechanism *locking with helping* or, interchangeably, *wait-free locking,* and I call the locks designed in this chapter *helping locks* or *wait-free locks.*

This section is organized as follows.

In Subsection 3.3.1, I describe a basic uniprocessor kernel mechanism that implements helping.

In Subsection 3.3.2, I extend the helping mechanism for multiprocessor kernels. I explain the execution model for multiprocessor helping; the problem of locking and waking up remote threads, and its solution; and the resource-access protocol implemented by multi-CPU helping, the multiprocessor priority-inheritance protocol (MPIP). I also compare MPIP to Rajkumar's and colleagues' multiprocessor priority-ceiling protocol.

Subsection 3.3.3 discusses wait-free locking in multithreaded user-mode programs. I propose wait-free lock designs for three different kernel interfaces. Also, as blocking must not occur for nonblocking synchronization to word, I discuss ways to avoid blocking in user-mode programs.

In Subsection 3.3.4, I discuss wait-free locking's restriction of disallowing voluntary blocking inside critical sections. I compare the expressive power of wait-free locking primitives to monitors, and I provide evidence that the restriction does not appear to be a limitation.

Finally, Subsection 3.3.5 reconsiders the serializer (or single-server) idiom that Massalin and Pu recommended for synchronizing accesses to complex data structures in lock-free operating systems [MP91]. I describe a simple modification of the serializer pattern to make it applicable in real-time systems, and I compare it to the wait-free lock designs developed in this chapter.

3.3.1 Wait-free locking for a uniprocessor kernel

I propose a wait-free locking-with-helping scheme. Each object to be synchronized in this fashion is protected by a lock with a "wait" stack, or more correctly, with a helper stack.

A lock knows which thread holds it upon entering a critical section protected by this lock. When a thread, *A*, wants to acquire a lock that is in use by a different thread, *B*, it puts itself on top of the lock's helper stack. Then, instead of blocking and waiting for *B* to finish, it helps *B* by passing the CPU to *B*, thereby effectively lending its priority to *B* and pushing *B* out of its critical section. Every time *A* is reactivated[2], it checks whether it now owns the lock; if it does not, it continues to help *B* until it does. When *B* finishes its critical section, it will find a helping thread on top of the lock's stack—in this case, thread *A*—and passes the lock (and the CPU) to that thread.

Using a stack instead of a FIFO wait queue has an important advantage: Given that threads are scheduled according to hard priorities, it follows that the thread with the highest priority lands on top of the helper stack. There is no way for a lower-priority thread to get in front of a higher-priority thread: As the thread running with high priority does not go to sleep after enqueuing in the helper stack, it cannot be preempted by a lower-priority thread and remains on top of the stack.[3] This property

[2]Reactivations of *A* can occur because the previous time slice has been consumed; a higher-priority thread became unrunnable; or a higher-priority thread started to help *A* to finish another critical section *A* had entered before.

[3]This is generally true only for uniprocessors. For multiprocessors, the priority ordering of the helper list could be ensured by using a different data structure—a priority queue—or by first migrating the helper to the CPU of the lock owner to force it into that CPU's priority-based execution order. There are subtle arguments for both designs, which I discuss in the next section (Section 3.3.2).

ensures that the highest-priority threads get their critical sections through first. It makes my locking mechanism an implementation of priority inheritance.

When *A* helps *B*, *B* inherits *A*'s current priority. If *B* wants to acquire another resource that is locked, it needs to enqueue in that resource's helper stack. It follows that threads in a helper stack are always ordered by *current* priority—independent of whether the threads actually own or have inherited that priority. As long as *A* keeps helping *B* (that is, until *B* releases the resource desired by *A*), *B* runs with *A*'s current priority. Therefore, the priority ordering is never violated (as long as critical sections are properly nested). Also, threads in a lock's helping stack always have a priority higher or equal to that of the lock owner.

Of course, execution of critical sections may be preempted by higher-priority threads that become ready to run in the meantime. However, to ensure wait-freedom, threads executing a critical section must not sleep or wait.

Instead, threads first must leave critical sections they have entered before they go to sleep. This requirement raises the question of how to deal with producer–consumer–like situations without race conditions. There are a number of textbook solutions for this problem. I describe my solution in Section 4.2.3.

3.3.2 Wait-free locking for a multiprocessor kernel

In this section, I extend the wait-free locking scheme of Section 3.3.1 to multiprocessor architectures. I present a resource-access protocol for tightly-coupled multiprocessors, which I call the multiprocessor priority-inheritance protocol (MPIP).

The design I present in this section has two desirable properties. First, it minimizes the number of inter-processor interrupts (IPIs) in the system. Second, for the normal (uncontented) case, it avoids synchronous inter-processor notifications (where one CPU needs to wait for the result of an IPI it sent to another CPU), thereby removing the effect of IPI latency on CPU-local execution—even when manipulating remote threads.

This section is organized as follows.

In Subsection 3.3.2.1, I give an overview over basic architectural assumptions and derive design principles for a multiprocessor kernel. In the following subsections, I use these design principles to develop a wait-free lock for multiprocessor systems. I explore design alternatives and explain the choices I have made for my lock.

In a kernel, one of the resources that need to be locked are threads running on remote CPUs. Locking remote threads raises the questions of dealing with those remote threads that currently execute on another CPU (lockdown) and with wakeups. I discuss these issues in Subsections 3.3.2.2 and 3.3.2.3. In Subsection 3.3.2.4, I discuss cross-CPU helping—the mechanism that implements MPIP and provides system-wide priority inheritance. Subsection 3.3.2.5 compares MPIP to MPCP.

I conclude this section in Subsection 3.3.2.6 with a summary.

3.3.2.1 Multiprocessor execution model

To benefit maximally from having multiple CPUs available, it is important to structure a system such that CPUs interact infrequently. I have designed Fiasco to minimize synchronization between CPUs using the following three design principles, which I discuss in turn:

- Prefer CPU-local data structures where possible.

- Run user threads only on their “home CPU,” that is, statically bind threads to one CPU.
- Manipulate remote threads locally.

CPU-LOCAL DATA STRUCTURES. Data structures that must only be accessed on a specific CPU are preferable in many cases because they do not cause cache contention and are very easy to synchronize.

One example for an important data structure to keep local are the ready queues. These queues keep track of all runnable threads in the system and are consulted whenever the kernel has to make a scheduling decision. In an IPC-intensive system such as a microkernel-based one, there are many context switches, and potentially many ready-queue accesses and updates. Context switches need to be very fast because of their frequency, and the efficiency of ready-queue accesses has a very direct influence on the efficiency of context switches.

STATIC CPU BINDING. Systems that dynamically schedule user-mode threads on multiple CPUs run danger of cache pollution: Cache working sets continuously have to be exchanged between CPUs, slowing down applications and increasing the likelihood of cache misses. That’s why the L4 philosophy is to bind threads to a specific CPU and let a (user-level) scheduler decide when to migrate a thread between CPUs.

I follow this belief in my wait-free lock design. I assume that threads are bound to a “home CPU,” and that migration only occurs on user request.

However, the lock implementation needs support for temporary remote execution of a thread’s in-kernel part. This feature facilitates helping, which I explain in Section 3.3.2.4. It does not induce more cache pollution than strict static binding because of two reasons: First, the kernel’s code runs on all CPUs and repeatedly reloads its working set into each CPU’s cache, that is, it always “pollutes” the cache. Second, this feature is used only when two threads interact, that is, when one thread locks another thread, which indicates that the locked thread’s kernel data is required on both threads’ CPUs.

MANIPULATING REMOTE THREADS LOCALLY. One of the resources that a kernel needs to manipulate are threads. When a thread *A* wants to lock down and manipulate another thread *B* running on a different CPU, there are two basic ways to implement their interaction:

Remote execution (i. e., local locking): Thread *A* runs the operation on *B*’s CPU. Thread *B* is locked on its own CPU.

Local execution (i. e., remote locking): Thread *A* runs the operation locally on its own CPU. Thread *B* is locked on *A*’s CPU.

Remote execution simplifies synchronization as all accesses to thread *B* are serialized on *B*’s CPU. However, it implies that thread *B*’s CPU needs to be notified using an expensive inter-processor interrupt (IPI) each time thread *B* is manipulated. Depending on the synchronous nature of the manipulation, another IPI may be necessary at the end of the operation. In addition to expensive notification, this solution is quite complex because it needs to deal with the following situation: When *A*’s IPI arrives on *B*’s CPU, thread *B* might have migrated to another CPU, so the IPI needs to be resent to that CPU.

Local execution, on the other hand, saves the costly notification if thread *B* is not runnable at the time it is locked—which is true in the majority of cases (synchronous IPC). Also, it avoids the thread-migration problem because threads can be locked on any CPU. However, a precondition for using this option is the availability of an inexpensive remote-locking primitive that prevents the thread from being scheduled.

Besides IPIs, one must also take into account caching effects. Remote execution ensures that only one particular CPU ever touches a thread's attributes, whereas local execution implies that critical sections on all CPUs can touch thread data, leading to cache-line invalidations and cache-line transfers between CPUs. However, consider that these transfers occur only when a thread *A* updates a remote thread *B*'s state. The updated data has to be transferred to *B*'s CPU's cache at some point regardless of which synchronization scheme is used. It follows that about the same number of cache-line transfers occur for both options, allowing me to exclude caching effects from further consideration.

My lock design assumes the second variant, local execution. Critical sections usually execute on the locker's CPU (i. e., except if helping occurs—see Section 3.3.2.4). In the uncontented case, remote locking can be implemented using a single compare-and-swap (CAS) operation. If the CAS fails (because the thread is currently running or because another thread owns the lock), the lock falls back to remote notification (to lock down a running thread) or helping (in case the thread is already locked). I explain these mechanisms in detail in Sections 3.3.2.2 and 3.3.2.4.

In my lock design, the locking primitives never need to *synchronously* notify the locked thread's CPU. Critical sections do not access CPU-local data structures directly (only unlocked code—code not executed in a critical section—does). Therefore, locked operations can run without notification overhead on any CPU. The only IPI that locked operations sometimes do generate is an *asynchronous* ready-queue–update notification when a locked thread becomes runnable. I describe the wakeup mechanism in detail in Section 3.3.2.3.

3.3.2.2 Lockdown

When a thread *A* wants to manipulate another thread *B* that is currently executing on another CPU, it must first cause *B* to stop running before it can proceed with its operation ("lockdown").

As I detailed in the previous subsection, I assume local execution of locked operations. That implies that locking of a remotely running thread *B* occurs on the locker's (*A*'s) CPU.

In the uncontented case (*B* is neither locked nor running), *A* can acquire *B*'s thread lock using a single atomic CAS operation. In that case, no IPI, spinning, or helping is necessary.

I propose that *A* locks down *B* after it has acquired *B*'s thread lock. Once *B* is locked, it cannot be activated on any CPU, nor can it migrate to any other CPU. However, it might still be running. When *A* detects that this is the case, it sends an IPI to *B*'s current execution CPU (which might not be *B*'s home CPU if *B* is being helped[4]; see Section 3.3.2.4). This IPI causes an immediate reschedule on *B*'s CPU. Meanwhile, *A* polls *B*'s status, waiting for *B* to be deactivated.

[4]For example, *B* has locked *D*; *C* also wants to lock *D* and helps *B* by lending it CPU time on *C*'s CPU.

Please note that while *A* is polling, waiting for *B* to stop running, *A* can still be preempted. This does not limit the throughput of operations that lock *B*, as another thread that wishes to lock *B* can help *A* to finish its critical section.

The lockdown operation requires additional synchronization to prevent deadlock when two threads try to lock down each other. I propose to secure the thread-lock operation using one simple (test-and-set) lock per thread. Thread *A* tries to acquire both its own and *B*'s lock before proceeding with the IPI. If sequentially acquiring both locks fails, a thread releases the locks and idles for a short amount of time, using a randomized exponential backoff, before it retries the operation.

3.3.2.3 Wakeup

When a locked operation wakes up the locked thread, the kernel must make a scheduling decision once the locked operation finishes: Should it run the previously locked thread immediately, or should it put the thread into the ready queue? On which CPU should the thread run, and on which CPU should the kernel carry out the enqueue operation?

In the uniprocessor case, the solution is very straightforward: In the thread-unlock operation, check whether the locked thread is runnable, and if so, switch to it if it has a higher priority; otherwise, enqueue it in the ready queue.

My multiprocessor solution is based on the CPU-local data structures and static CPU binding principles: I propose that the kernel never runs kernel code not executing a critical section (or user code) on a CPU other than a thread's home CPU. Instead, the thread-unlock operation queues the thread in its home CPU's wakeup queue and asynchronously notifies that CPU using an IPI. When a CPU receives this notification, it enqueues the thread in its ready queue, or—if the thread has the highest priority—directly switches to the thread.

3.3.2.4 Multiprocessor priority-inheritance protocol (MPIP)

As the uniprocessor lock I have described in Section 3.3.1, the multiprocessor lock implements priority inheritance using helping.

Priority inheritance is desirable even across CPU boundaries to avoid situations in which a thread on one CPU prevents the highest-priority thread on another CPU from making progress. Therefore, I explored ways to provide a helping mechanism that works in a multiprocessor environment.

Cross-CPU helping occurs when a thread *A* on one CPU wants to acquire a lock held by a thread *B* on another CPU. There are two basic variants for implementing helping:

Remote helping: Helping occurs on thread *B*'s CPU. Thread *A* migrates to *B*'s CPU. If its priority is higher than that of a currently running thread on that CPU, it can lend the priority to *B*.

Local helping: Helping occurs on thread *A*'s CPU. Thread *B* temporarily runs on *A*'s CPU (the *helper CPU*) for the duration of its critical section.

These two variants have slightly different semantics: With local helping, it is possible that thread *A* helps a thread *B* that has a *higher* priority, but is blocked on its home CPU by a thread with an even higher priority. With remote helping, thread *A* would be put to sleep in this case, and no helping at all would occur. In consequence, remote helping requires a global, "end-to-end" priority scheme to prevent priority inversion.

Local helping has the drawback that it slightly softens the strict processor binding of threads. However, this is true only for kernel code, because only the kernel's critical sections temporarily run on a helper CPU; the effect is not visible on user level. The advantage of local helping is that it allows the system to remain wait-free: It ensures that *on each CPU* the highest-priority thread can make progress; that thread cannot be blocked by remote threads. Therefore, I prefer local helping's behavior.

Apart from semantics, remote helping is less preferable also because it is more complex to implement and has a higher run-time cost: Like remote locking (see Section 3.3.2.1), it must deal with the thread-migration problem: At the time thread *A* arrives at *B*'s CPU, thread *B* might have migrated elsewhere. Also, in case *A* in not the highest-priority thread on *B*'s CPU, it must be added to that CPU's ready queue.

On the other hand, local helping is a low-overhead operation. During helping, no cross-CPU synchronization is needed; the helping thread just passes the CPU to the current lock owner. Also, this operation does not require a remote ready-queue update: The remote, lock-holding thread is runnable per definition (lock owners are not allowed to sleep, as that would violate the nonblocking predicate), but not running. It follows that it is already enqueued in its home CPU's ready queue. The helping thread executes only locally on its home CPU, so the normal CPU-local lazy-queueing discipline applies.

For these reasons, I propose local helping.

In the remainder of this section, I will discuss two design issues that arise with local helping: Behavior when the current lock owner actually executes on some CPU, and scheduling after helping.

"HELPING" A RUNNING THREAD. What happens if a thread *A*, intending to help another thread *B*, finds that *B* is already running on another CPU? I considered two alternatives:

Sleep and callback. Thread *A* registers a callback IPI with thread *B*'s current CPU and goes to sleep, allowing other threads to run. As soon as *B* finishes its critical section or stops running, *B* sends an IPI to *A*'s CPU (and all other helpers' CPUs), waking *A* (and all other helpers) up again.

Polling. Thread *A* does nothing except polling *B*'s thread state and the lock's state, waiting for *B* to stop running or leaving the critical section.

Synchronization cost and latency are higher with the callback method: It requires extra checks in the unlock and thread-deactivation code paths and an IPI to wake up helpers. However, the callback method can result in higher CPU utilization as other threads can run while a thread *A* is waiting for thread *B* to finish, whereas the polling method potentially burns a whole time slice doing nothing. Yet, this danger does not contradict real-time principles (the critical section delaying the high-priority thread *does* execute), nor is it very probable given that critical sections usually only execute for a fraction of a time slice.

Therefore, I propose to use the polling method.

There is a fixed order in which helping (or polling) threads acquire a lock: As with the uniprocessor version of helping, helpers enqueue in the lock's "helper queue," which is sorted by global priority[5], and a thread that releases the lock transfers lock ownership to the highest-priority helper. Consequently, low-priority threads cannot starve high-priority threads from accessing the lock.

SCHEDULING AFTER HELPING. When a thread *B* has been helped and has executed its critical section on a helper CPU, which thread should run on that CPU once *B* leaves its critical section?

If thread *B* was helped, there is at least one other thread that wants to acquire the lock (the helper). This implies that there always is a new lock owner after *B* leaves its critical section. This is the highest-priority thread that was waiting for the lock. It can be equivalent with thread *B*'s helper, but this need not be the case if there is a higher-priority thread waiting for the lock on another CPU, polling. It follows that *B* cannot unconditionally switch to the new lock owner—provided this was desirable—as that thread might already run on another CPU.

The static CPU binding principle mandates that unlocked code and user code only run on a thread's home CPU. In other words, staying on its helper CPU is also not an option for thread *B*.

Thread *B* could switch to its helper, but that would require keeping track of the current helper, and is ambiguous if more than one thread helped *B*.

The only option is for thread *B* to call the scheduler. The scheduler will select the highest-priority thread whose home CPU is *B*'s current helper CPU. It will never select thread *B* again, independent of *B*'s priority, because *B* has a different home CPU.

Once thread *B* has been descheduled from its helper CPU, it becomes runnable on its home CPU again. No special notification is necessary: *B* was enqueued in its home CPU's ready queue already before it was helped (because at that time, it was runnable, but not executing), which means that the scheduler can consider it automatically. Also, if *B* now is the highest-priority thread of its home CPU, that CPU's scheduler can poll, waiting for *B* being removed from helper CPUs, and run it immediately.

3.3.2.5 Comparison MPIP–MPCP

In comparison to MPCP, MPIP uses a radically different multiprocessor execution model.

- In MPCP, resources (and critical sections accessing resources) are bound to fixed CPUs. In MPIP, critical sections can execute on all CPUs. In the common (uncontented) case, threads never migrate under MPIP.
- While MPCP disallows nested critical sections (i. e., it disallows locking more than one resource), there is no such restriction in MPIP.
- There are no priority ceilings for resources in MPIP. Therefore, MPIP does not prevent deadlocks or chained blocking.
- In MPIP, all threads finish their critical sections in a finite number of steps. Resource-contention blocking is bounded for blocking induced by both local and remote threads. Therefore, MPIP is a wait-free synchronization protocol.

[5]On multiprocessor systems, the "helper stack" of Section 3.3.1 cannot be used, because the stack entries would not be automatically ordered by priority as on uniprocessors. Therefore, an explicitly priority-sorted queue is needed.

In this work, I do not look into schedulability analysis for MPIP—such analysis is outside the scope of this thesis. However, as MPIP is wait-free, existing analysis methods for wait-free synchronization protocols can be applied.

3.3.2.6 Summary

In this section, I developed a wait-free lock with helping for multiprocessor systems. My lock design implements a novel resource-access protocol, multiprocessor priority inheritance.

I started from three design principles: CPU-local data structures, static CPU binding, and manipulating (locking) remote threads locally.

From these principles, I derived a remote-locking design with the following properties: In the uncontented case, remote-thread lockdown does not require an IPI. It is never necessary to access the ready queue of a remote CPU. Kernel code running within a thread context always runs on the CPU to which the thread is bound, except in the case of helping: When a thread helps another thread to finish a critical section, the helped thread can execute on the helper's CPU for the duration of its critical section. After a thread was helped on a remote CPU, it always releases that CPU, waiting for its home CPU's scheduler to pick it up again. Helping does not occur when the thread blocking a critical section is currently executing on another CPU; in that case, the thread that wishes to enter its critical section simply waits, polling the lock's state.

In combination, these properties minimize the number of IPIs. For the normal (uncontented) case, they completely eliminate the need for synchronous notifications where one CPU needs to wait for the result of an IPI it sent to another CPU; in that case, IPI latency therefore has no effect on CPU-local execution, even for remote-thread manipulation.

3.3.3 Wait-free locking in user-mode programs

If a kernel provides static priorities or an implementation of priority inheritance, it is possible to implement the wait-free locking synchronization scheme of Section 3.3.1 in user mode. In this section, I consider three different kernel interfaces that allow multithreaded applications to take advantage of priority inheritance, and I show how to implement a wait-free lock without race conditions on top of them in user mode.

Lock with priority inheritance is an interface to a lock implementation with priority inheritance in the kernel.

Sleep and wakeup with priority inheritance is an interface to a sleep–wakeup mechanism or another IPC mechanism that implements priority inheritance. If a kernel offers such an interface, it is possible to construct a user-level implementation of a lock with priority inheritance.

Time-slice donation is a means of temporarily yielding the CPU to a different runnable thread. When a kernel offers a time-slice–donation interface together with hard priorities, but not a direct interface to an implementation of priority inheritance, it is possible to emulate priority inheritance at user level.

In the following three subsections, I describe possible implementations (for both the kernel-space and the user-space parts) for locking with priority inheritance based on the preceding kernel interfaces. These implementations are meant for multithreaded applications in which threads share state in common memory. Likewise, my lock designs also store their state used for synchronization in shared memory—with the exception of the first design, which keeps all state in the kernel.

In Subsection 3.3.3.4, I discuss the problem of avoiding blocking in user-space programs—a precondition to implementing wait-free synchronization.

In a final subsection (Section 3.3.3.5), I summarize and compare the interface options, and, based on my analysis, I propose a modification of the L4 microkernel interface.

3.3.3.1 Lock with priority inheritance

In this case, the kernel exports an interface like this:

```
Lock_handle new_prio_inherit_lock ();
void        prio_inherit_lock (Lock_handle l);
void        prio_inherit_unlock (Lock_handle l);
```

If a kernel implements a wait-free lock with priority inheritance such as the one I described in Section 3.3.1, it is trivial to export it to user space. The kernel must be able to dynamically instantiate wait-free locks, and it needs to export the lock's operations to user space using a lock handle.

With locks implemented as kernel objects, there is the issue of access control. Handles are capabilities the kernel must protect by some means in order to prevent malicious or buggy programs from manipulating locks they should not have access to. There are a number of textbook solutions to this problem, including the following solutions:

Additional level of indirection. Each thread has its own handle name space, and there are additional system calls to transfer access rights between threads. This is the way Unix protects file handles.

Cryptographic capabilities. Locks are shared by passing around the handle's ID directly. Anyone who knows a handle can access the lock. However, it is difficult to guess handles because they are encoded in a large number.

One problem with the kernel-object approach is its cost. User programs must use system calls to lock and unlock objects, independent of whether there is or is not contention. The solutions I develop in the next two subsections do not require entering the kernel when there is no contention.

3.3.3.2 Sleep and wakeup with priority inheritance

In this case, the kernel exports an interface like this:

```
void thread_sleep (Threadid wait_for);
void thread_wakeup (Threadid sleeping);
void thread_wakeup_any ();
```

The `thread_sleep()` operation puts the thread to sleep, waiting for a wakeup from a specific other thread. This call also implements priority inheritance: The thread named in the `wait_for` argument inherits the priority of the sleeping thread. The `thread_wakeup()` operation wakes up a specific thread, and `thread_wakeup_any()` unblocks exactly one thread waiting for the current thread—the one with the highest priority.[6]

This interface is equivalent to a synchronous IPC interface such as the one in the L4 API: The `thread_sleep()` primitive is equivalent to an IPC-send operation, `thread_wakeup()` is equivalent to a IPC-receive operation and unblocks the named sender, and `thread_wakeup_any()` is equivalent to an "open" IPC-wait.

A variation of this interface uses a kernel-implemented wait queue named with a handle.

KERNEL SUPPORT. I assume that both the sleep (or IPC-send) and specific wakeup (or IPC-receive) operations implement priority inheritance and put sleepers to an internal wait queue in priority order. I further assume that these operations are synchronous, that is, wakeup blocks until there is a thread to wake up.

To implement priority inheritance using sleep and wakeup, a kernel needs to keep track of wakeup dependencies. A thread *A* depends on a wakeup from another thread *B* if *A* is suspended in a `thread_sleep()`, waiting for a wakeup from *B*. I see two different ways of storing these dependencies: in memory local to sleeping threads, and in global memory accessible by the scheduler.

In both designs, the kernel does not remove threads that called `thread_sleep()` from the ready queue. In the local-data design, whenever a sleeping thread is scheduled or activated, it immediately switches to the thread it depends on for its wakeup. In the global-data design, the thread-switching code consults a thread's dependencies before switching to it. The latter variant avoids unnecessary context switches, but adds some overhead to the thread-switching code.

In any case, keeping track of wakeup dependencies raises the problem of cycle detection. If two or more threads depend on each other in a cycle, the kernel needs a way to detect that it has traversed a thread already in the current dependency-resolution pass; otherwise dependency resolution does not terminate. A simple solution is to use a global version number that the kernel increases on every scheduling decision. When the kernel passes a thread during dependency traversal, it updates that thread's local version number. If the thread's version is already equivalent to the global version number, dependency resolution has failed, and the traversed threads need to be removed from the kernel's ready queue.

USER-MODE LOCK IMPLEMENTATION. The basic problem with this set of interfaces is that a thread waiting for the lock does not know in advance which other thread will wake it up. On the other hand, the kernel needs to know exactly which thread is expected to send the wakeup notification.

I propose two different solution to this problem, each with its own performance–resource usage tradeoff. Both solutions are cooperative solutions, that is, they require the cooperation of all participating threads.

Figure 3.1 shows pseudocode for my first solution. This design uses an extra thread for access mediation. Threads that want to acquire a lock indicate this in a counter variable and then go to sleep waiting for the mediator (using `thread_sleep()`). The mediator only runs when it inherits a priority—otherwise it is suspended (in L4 terms: it has a priority of 0). Once a thread waits for it, it starts up and goes to sleep itself

[6] Contrary to the wakeup operation known from condition variables, `thread_wakeup()` and `thread_wakeup_any()` are synchronous and block, waiting for a thread to wake up.

```
class User_mutex1
{
  Threadid d_mediator;
  Threadid d_owner;
  int      d_waiting;

public:
  void lock ()
  {
    while (! CAS (&d_owner, 0, current ()))
      {
        ++d_waiting;
        thread_sleep (d_mediator);
      }
  }

  void unlock ()
  {
    if (d_waiting)
      thread_wakeup (d_mediator);
    else
      d_owner = 0;
  }

  void mediator_thread ()
  {
    for (;;)
      {
        // We woke up -- we inherited someone's priority.
        // Pass on priority to current lock owner.
        Threadid owner = d_owner;
        if (owner)
          {
            thread_sleep (owner);
            --d_waiting;
            d_owner = 0;
          }
        else
          --d_waiting;

        thread_wakeup_any ();
        // Highest-priority waiter starts up, grabs lock.
        // We go to sleep again
        // (unless there is waiter of equal priority).
      }
  }
};
```

Figure 3.1: User-mode lock with priority inheritance using sleep–wakeup interface, using a mediator thread

waiting for the current lock owner. The lock owner checks the counter variable before releasing the lock. If the counter is nonzero, the lock owner wakes up the mediator using `thread_wakeup()`. The mediator wakes up the highest-priority waiting thread using `thread_wakeup_any()`.

Provided that it is possible to ensure that the kernel never schedules the mediator thread unless that thread inherits some other thread's priority, this implementation is race free.

Figure 3.2 illustrates my second solution, which is especially suited for uniprocessors. Here, there is no mediator thread, and waiting threads lend their priority directly to the current lock holder. However, when a lock holder releases the lock, it needs to explicitly wake up *all* waiters. This is necessary because the waiting threads need to update their priority-inheritance relationship. My implementation in Figure 3.2 uses a counter to keep track of the number of waiters to wake up. The thread doing the wakeup in `unlock()` only knows the number of threads to wake up, not their identity, so it uses the unspecific wakeup operation `thread_wakeup_any()`, relying on that primitive to wake up waiters (which, in IPC terms, are *senders* waiting in the lock owner's send queue) in priority order. If the thread cannot use `thread_wakeup_any()` because it has other IPC relationships that could interfere, a variant of this solution uses a queue of sleeping threads instead of a counter. In this variant, `unlock()` is responsible for waking up the threads on the queue on priority order after it has detached the queue from the lock with the CASW.[7] I have not shown this variant in Figure 3.2, but next section's switching lock (shown in Figure 3.3) uses this idea.

As the `unlock()` operation unconditionally wakes up all threads that have blocked during the critical section, the following question arises: Do the threads woken up by `unlock()` become runnable all at once, compete for the lock, and (except for the winner) end up calling `thread_sleep()` again? Such behavior is considered a problem because it would result in excessive overhead. It is known as the *thundering-herd problem.*

On uniprocessors, this lock design does not suffer from the thundering-herd problem despite the need to wake up all waiters, provided all threads differ in priority. Once the highest-priority thread has been woken up, it restarts and grabs the lock. The previous lock owner cannot restart more threads until the high-priority thread stops. When that happens, the previous lock owner restarts the thread with the next-highest priority, and so on. Therefore, each sleeping thread gets restarted exactly once in this case.

On multiprocessors, the thundering-herd problem only can be circumvented by binding all participating threads to a single CPU. Therefore, this mechanism is not practical for synchronizing accesses to global data.

3.3.3.3 Time-slice donation

In this case, the kernel exports an interface like this:

```
void thread_switch (Threadid other);
```

This system call passes the current time slice to another thread, independent of that thread's priority. The call simply does nothing if the other thread is not ready to run.[8]

[7] Recall that CASW is the double-word version of CAS—see Section 2.2.1.2.

[8] The L4 system call *thread_switch* currently does a full reschedule in this case and would have to be extended to support these semantics.

```
class User_mutex2
{
  struct User_mutex_data {
    Threadid owner;
    int      waiting;
  };

  static const User_mutex_data nil = { .owner = 0, .waiting = 0 };

  User_mutex_data d_data;

public:
  void lock ()
  {
    for (;;)
      {
        User_mutex_data myself
          = { .owner = current(), .waiting = 0 };

        if (CASW (&d_data, nil, myself))
          break;                // Success

        User_mutex_data state_new, state_old;

        state_old = state_new = d_data;
        ++state_new.data.waiting;

        if (! CASW (&d_data, state_old, state_new))
          continue;             // Retry

        thread_sleep (state_new.owner);
      }
  }

  void unlock ()
  {
    int wait_count;
    User_mutex_data state_old;

    do
      {
        state_old = d_data;
        wait_count = state_old.waiting;
      }
    while (! CASW (&d_data, state_old, nil));

    while (wait_count--)
      thread_wakeup_any ();
  }
};
```

Figure 3.2: User-mode lock with priority inheritance using sleep–wakeup interface, *without* a mediator thread

KERNEL SUPPORT. When implementing priority inheritance at user level using this interface, the only requirement on the kernel is that it schedules threads according to static priorities. In particular, it is not necessary that the kernel implements priority inheritance itself.

USER-MODE LOCK IMPLEMENTATION. The user-level implementation of locking with priority inheritance using a thread-switching operation can be very similar to the in-kernel implementation I described in Section 3.3.1. However, the user-level implementation needs some extra care to avoid race conditions. A kernel can easily combine checking of lock ownership, queuing in the lock's queue, and switching to the lock owner into one atomic operation. Without special kernel support, this is impossible at user level.

Figure 3.3 shows a possible implementation. It picks up the wait-queue idea from the end of previous section.

Like the in-kernel design I proposed in Section 3.3.1, this implementation uses a stack (`d_data.waiting`) to keep track of threads that are busy helping. However, there are also a number of differences to the in-kernel design.

First, the helping threads are notified explicitly whenever the lock owner releases the lock. When that happens, helping threads retry an atomic lock acquisition. In contrast, the in-kernel variant's unlock operation directly transfers lock ownership to the highest-priority waiter. However, it implies that the next owner can be atomically dequeued from a live (i. e., potentially being modified from another thread), synchronized stack or list—which is difficult in user mode.

Second, I do not assume the stack of helping threads to be in priority order. Instead, my `unlock` implementation searches for the highest-priority thread in the stack before waking it up. This makes the design more applicable for multiprocessors. It is possible to search the stack because it is not live and does not need to be synchronized anymore—only the thread doing `unlock` has access to it.

This implementation is subject to the same thundering-herd issues as previous section's second implementation. Please find a discussion of these issues at the end of that section. In essence, it works best if all participating threads run on a single CPU.

3.3.3.4 Avoiding blocking in user-mode programs

To avoid blocking inside critical sections, user programs must take extra care typically unnecessary in the kernel: They need to ensure that critical sections do not trigger page faults leading to blocking. For that, user programs need operating-system support.

I consider three possible operating-system interfaces to deal with this problem:

Memory pinning. The trivial solution is to prevent page faults. To make this possible, the operating system has to provide an interface for memory pinning.

Nonblocking memory manager. Paging that leads to blocking (such as page-ins from disk) is forbidden, but page faults that can be resolved without blocking (such as a copy-on-write implementation) are allowed. In this case, high-priority threads must trust the memory managers of low-priority threads with whom they share data. As a consequence, the operating system must implement priority inheritance in its page-fault protocol.

Recover from page faults. This solution works by letting the page-fault handler know that the user program is executing a critical section. As for a nonblocking memory manager, a precondition to using this method is that the page-fault handler inherits the priority of the faulting thread.

The page-fault handler usually runs inside the kernel. However, microkernels such as L4 provide user-level page-fault handling. For successful user-mode handling of page faults, it is necessary to ensure that the user-level page-fault handler does not generate a blocking page fault itself. This can be accomplished by combining this method with either of the first two variants (memory pinning; or trusted nonblocking memory manager).

The page-fault–recovery method allows the page-fault handler to detect that a page fault occurred while executing a critical section, and to initiate some form of recovery that prevents the critical section from blocking (such as rolling back the critical section). Page-fault recovery has the advantage over the nonblocking–memory-manager strategy that it does not depend on cooperation of the memory manager.

Let me discuss two possible forms of recovery:

Recovery in the page-fault handler. If the page fault interrupted a critical section, it does any recovery necessary before its usual processing.

In systems that allow user programs to define their own page-fault handler (such as L4-based systems), this is a very flexible solution as user programs can define an application-specific recovery strategy.

With a hard-wired kernel page-fault handler, this solution is not as flexible. The page-fault handler can roll back the critical section to its beginning and free the lock. This method assumes that the page-fault handler knows the implementation of the critical section, or that there is an interface that informs the page-fault handler about the steps that need to be undone. This is certainly possible, but it defeats the ease of use of the synchronization schemes I presented in previous sections.

Reflect a trap to the user code. In this case, the page-fault handler sends the faulting user code a signal. The user code performs the recovery itself in any way it wants. It can abort or roll-back the critical section, or it can communicate with a memory manager using a synchronization scheme that provides priority inheritance.

If page-fault handling is possible in user mode, it is possible to implement the second form of recovery in terms of the first: When the page-fault handler detects that a page fault occurred in a critical section, it can deliver a signal to the faulting thread (using a system call such as L4's *lthread_ex_regs*).

User-mode recovery combined with either or both of memory pinning and nonblocking memory manager is the most flexible solution. The combination of memory pinning and page-fault reflection has the nice property that it is very easy to provide in any operating-system kernel.

3.3.3.5 Summary

Multithreaded user programs can use my design technique if the operating system provides some support that real-time systems provide frequently, or can easily implement:

```
class User_mutex3
{
  struct Waiting {
    Threadid thread;
    int      priority;
    Waiting* next;
    bool     queued;
  };

  struct User_mutex_data {
    Threadid owner;
    Waiting* waiting;
  };

  static const User_mutex_data nil = { .owner = 0, .waiting = 0 };

  User_mutex_data d_data;

public:
  void lock ()
  {
    for (;;)
      {
        User_mutex_data myself
          = { .owner = current(), .waiting = 0 };

        if (CASW (&d_data, nil, myself))
          break;                // Success

        User_mutex_data state_new, state_old;
        Waiting w;

        state_old = state_new = d_data;
        state_new.waiting = w;
        w.thread   = current ();
        w.priority = current_priority ();
        w.next     = state_old.waiting;
        w.queued   = true;

        if (! CASW (&d_data, state_old, state_new))
          continue;             // Retry

        // Keep helping until we are dequeued
        while (w.queued)
          thread_switch (state_new.owner);
      }
  }
```

*** Continued on next page ***

Figure 3.3: User-mode lock with priority inheritance using switch interface

```
  void unlock ()
  {
    Waiting* wait_queue;
    User_mutex_data state_old;

    // Unlock
    do
      {
        state_old = d_data;
        wait_queue = state_old.waiting;
      }
    while (! CASW (&d_data, state_old, nil));

    while (wait_queue)
      {
        Waiting** highest_prio = &wait_queue;

        // Find thread with highest priority in queue
        for (Waiting** w = &(wait_queue->next);
             *w;
             w = &((*w)->next))
          {
            if ((*w)->priority > (*highest_prio)->priority)
              highest_prio = w;
          }

        Waiting* waking_up       = *highest_prio;
        Threadid waking_thread   = waking_up->thread;
        int      waking_priority = waking_up->priority;

        *highest_prio = waiting->next; // Dequeue
        waking_up->queued = false;

        if (waking_priority > current_priority())
          thread_switch (waking_thread);
      }
  }
}; // User_mutex3
```

Figure 3.3: (cont.) User-mode lock with priority inheritance using switch interface

MWCAS support or preemption-safe locking; priority inheritance; and an interface that allows the prevention of page faults that block the program.

I proposed three different kernel interfaces that enable a user-mode program to implement priority inheritance: Lock with priority inheritance, sleep and wakeup with priority inheritance, and time-slice donation. The first interface exposes a lock implemented as a kernel object, the second is based on a sleep-and-wakeup–based protocol such as IPC, and the third uses only thread switching. The first two interfaces assume that the kernel implements priority inheritance; the latter interface does not.

I discussed three kernel interfaces that allow a user program to cope with page faults that lead to blocking: memory pinning, nonblocking memory manager, and user-mode page-fault recovery combined with either or both of the first two.

For a second-generation microkernel that provides synchronous IPC—such as L4 implementations—, the IPC-based priority-inheritance protocol is the most natural one.

L4 microkernels support external user-mode memory managers, and the page-fault protocol for these memory managers is based on IPC. This makes priority inheritance in the page-fault protocol a natural extension. Also, memory pinning can be implemented using the standard L4 memory-management services. These features allow the use of all three of the blocking–from–page-fault avoidance techniques I proposed.

For kernels that map page fault to IPC (such as L4-based kernels like Fiasco), I therefore propose implementing IPC with priority inheritance. This feature enables both nonblocking memory management and the sleep–wakeup–based user-mode lock implementation of Section 3.3.3.2.

3.3.4 Strength of the wait-free locking

As long as critical sections do not nest, it is easy to see that my construction can be used to implement wait-and-notify monitors[9] [LR80] (or their recent descendant, Java synchronized methods). Whenever a monitor-protected object's method is called, the object's lock is acquired. The monitor's wait operation would then be implemented as an unlock–sleep–lock sequence. Figure 3.5 shows a possible monitor implementation that uses a simple lock-free semaphore, shown in Figure 3.4.

Synchronization is more difficult when more than one object can be locked at a time. I will discuss two scenarios: nested monitor calls (i. e., nested critical sections), and atomic acquisition of multiple locks.

As long as monitor methods never wait for events, locking with helping works for nested monitor calls in the same way as for non-nested monitors. However, if a nested method wants to wait for an event, freeing the nested monitor does not help because the outer monitor would still be locked during the sleep—which is illegal under my scheme. That is why nested monitor calls must not sleep.

There are two ways to deal with this restriction: Either construct the system such that second-level monitors or even all monitors never sleep, or make the locking more coarse-grained so that all objects that would have to be locked before going to sleep are in fact protected by a single monitor.

In the Fiasco microkernel, I have chosen the first option; in fact, I constructed the kernel so that critical sections never need to sleep. I discuss synchronization in the Fiasco microkernel in more detail in Section 4.2.3.

[9]There is a large variety of monitors with differing semantics, but most of them can be shown to have equivalent expressive power [How76, BF95]. Wait-and-notify monitors, also classified as "no-priority nonblocking monitors" [BF95], have first been used in Mesa [LR80].

A different situation arises if the locks a critical section needs are known before the critical section starts, and during its execution. In this case, the wait operation can release all locks before sleeping, and reacquire them afterwards. However, this method cannot maintain internal state while waiting for an event—the object has to be transferred into a consistent state before sleeping.

3.3.5 Real-time serializer

Before I implemented the wait-free locking scheme described in Section 4.2.3, I experimented with Massalin's and Pu's single-server synchronization scheme discussed in Section 2.2.1 [MP91]. In this section, I discuss how the single-server mechanism can be changed for real-time systems, and why I changed it into the simpler locking-with-helping scheme.

In Massalin's and Pu's scheme, threads that want to change an object put a change-request message into the request queue of the server thread that owns the object. In similar spirit to my helping-lock design from Section 3.3, I can minimize the worst-case wait time for high-priority threads by replacing the request queue with a stack (so that messages from high-priority senders get processed first), and by letting requesters actively donate CPU time to the server thread until it has handled their request.

When I first designed and implemented my wait-free synchronization mechanism, I drew inspiration from Massalin's and Pu's work. In particular, my design looked as follows:

My kernel ensured serialization of critical sections by allowing only one thread, an object's *owner*, to execute operations on that object. In other words, all locked operations ran in the thread context of the owner of an object.

Threads were their own owners. Consequently, threads carried out themselves all locked operations on them, including those initiated by other threads.

The kernel assigned ownership for other objects (not threads) on the fly using lock-free synchronization. This design can also be viewed as follows: The only object type that can be locked at all is the thread. All other objects are "locked" by locking a thread and assigning ownership of the object to that thread. Then, all operations on that object are carried out by the owner.

Helping an owner was as simple as repeatedly switching to the owner until either the owner had completed the request, or a thread that deleted the owner had aborted the request. The context-switching code took care of executing all requests before returning to the context of the thread.

I consider this design to be not inelegant, but unfortunately, it required a context switch for every locked operation. Only later I realized that this mechanism in fact shares many properties with the wait-free locking scheme with priority inheritance I derived in Section 3.3.1. My new locking mechanism is less complex and performs much better than my original single-server scheme.

```
class Binary_semaphore
{
  List<Thread*> d_q; // Lock-free thread list
  int           d_count;

public:
  void down ()
  {
    d_q.enqueue (current());

    int old;
    do
      {
        old = d_count;
      }
    while (! CAS (&d_count, old, old - 1));

    if (old > 0)
      {
        // Own the semaphore,
        // can safely dequeue myself
        d_q.dequeue (current());
      } else {
        sleep (Thread_sem_wakeup);
        // Have been dequeued in up ()
      }
  }

  void up ()
  {
    int old;
    do
      {
        old = d_count;
      }
    while (! CAS (&d_count, old, old + 1));

    if (old < 0)
      {
        Thread* t = d_q.dequeue_first ();
        wakeup (t, Thread_sem_wakeup);
      }
  }
}; // Binary_semaphore
```

Figure 3.4: Pseudocode for a simple lock-free binary semaphore (for single-CPU machines). It makes use of a lock-free list of threads (`Thread_list`) with a given queuing discipline, for example a FIFO queue or a priority queue, and `sleep` and `wakeup` primitives like those in Figure 4.1 (page 54).

```
class Monitor
{
  Helping_lock d_lock;

public:
  void enter ()
  {
    d_lock.lock (); // Locking w/ helping
  }

  void leave ()
  {
    d_lock.unlock ();
  }

  void wait (Binary_semaphore* condition)
  {
    d_lock.unlock ();
    condition->down ();
    d_lock.lock (); // Locking w/ helping
  }

  void signal (Binary_semaphore* condition)
  {
    condition->up ();
  }
}; // Monitor
```

Figure 3.5: Pseudocode for a wait-and-notify monitor based on a helping lock. This is a simple textbook implementation—except that it uses only nonblocking synchronization primitives. Semaphores used as condition variables need to be initialized with 0.

The `signal` operation wakes up a waiter according to the semaphore's queueing discipline. When one or more waiters have been restarted, and more threads are trying to enter the monitor, the `Helping_lock`'s helper stack guarantees that the thread with the highest priority can enter the monitor first.

Chapter 4

The Fiasco microkernel

I developed the Fiasco microkernel as the basis of the DROPS operating-system project—a research project exploring various aspects of hard and soft real-time systems and multimedia applications on standard PC hardware [HBB$^+$98]. The microkernel runs on uniprocessor and multiprocessor x86 PCs, and it is an implementation of the L4/x86 binary interface [Lie95]. It is able to run L^{4}Linux [HHL$^+$97], a Linux server running as a user-level program that is binary compatible with standard Linux, and it is freely available from `http://os.inf.tu-dresden.de/fiasco/`.

In this chapter, I concentrate on aspects of Fiasco that relate to the main subject of this thesis, pragmatic nonblocking synchronization. I have described other aspects of the system in a technical report that accompanies this thesis, "The Fiasco kernel: System architecture" [Hoh02].

This chapter is organized as follows.

In Section 4.1, I introduce Fiasco by way of defining its functional and real-time requirements. It will become apparent that Fiasco's requirements match the goal of the development methodology I proposed in Chapter 3—which should not come as a surprise as I developed the methodology from the kernel's requirements.

In Section 4.2, I describe in detail how synchronization works in Fiasco. I map Fiasco's kernel objects to the synchronization approaches of Chapter 3. I describe Fiasco's implementation of wait-free locking, and, as a case study, I look in detail at the synchronization of IPC in Fiasco.

4.1 Requirements

I have laid down the requirements for Fiasco in a technical specification [Hoh98].

I intended Fiasco to be a drop-in replacement for L4/x86. Therefore, Fiasco's specification includes verbatim the L4 Reference Manual [Lie96a], which includes a general description of the user-visible concepts of the L4 microkernel interface as well as a definition of L4's ABI for x86 CPUs.

In addition to being an implementation of L4, there were a number of other requirements for Fiasco. In this section, I first outline the general functional requirements imposed by the L4 programming interface, and then discuss other goals.

4.1.1 Functional requirements

The L4 programming interface (or short, L4) is a very minimal microkernel interface. As a general rule, L4 kernels only implement what *must* be implemented in kernel mode. Policies such as memory management or device drivers do not belong into the kernel and can be implemented at user level; L4 kernels only implement mechanisms [Lie95].

L4 uses two main user-visible abstractions: *address spaces* and *threads*. *Tasks* consist of an address space and at least one thread. Threads are active entities that execute user code (and kernel services on behalf of the user program).

Threads can communicate using *inter-process communication (IPC).* Threads can either store messages in buffers in their virtual memory, in which case the IPC is called a *long IPC.* There is also a fast IPC mode, *short IPC,* which does not copy a memory buffer, but only transfers register contents. L4 IPC is synchronous: Senders block until receivers actively request a message—that is, messages are not queued in the kernel, but senders are. To enable fast synchronous message transfer and fast RPC round trips, L4's IPC semantics allow an implicit context switch from sender to receiver, independent of these threads' priorities. Fiasco's IPC system call allows to disable this behavior to facilitate strict priority-based scheduling for real-time systems.

Address spaces contain memory references *(mappings)* and references to input–output ports *(I/O mappings)* a task's threads may access. Threads can pass along mappings using IPC. This special IPC data type is called a *flexpage.*

The kernel translates a thread *A*'s page faults into IPC messages *(page-fault messages).* It sends these messages on *A*'s behalf to a predestined thread *B* that is acting as *A*'s *pager*. Together with the flexpage-IPC mechanism, it is possible to implement any paging policy at user level.

The kernel also translates hardware interrupts (IRQs) to IPC messages *(IRQ messages).* Threads can attach to hardware-interrupt sources to register for IRQ messages. (Attaching to IRQs also uses a special IPC protocol.) Using this facility, interrupt-driven device drivers can be implemented as user-level threads.

Other traps and exceptions must be handled by the thread that causes them. L4 allows threads to install a thread-specific interrupt-descriptor table in x86 format using x86's `lidt` instruction (which must be emulated by the kernel). This facility allows the emulation of a different ABI; for example, L^{4}Linux uses it to emulate Linux' system-call ABI.

The kernel by default uses a built-in *priority-based round-robin scheduler* to multiplex threads on each CPU. This scheduler uses two thread attributes as parameters: *priority* and *time-slice length.* There are two other thread attributes that are meant to facilitate user-level scheduler implementations: *internal and external preemptor*. Preemptors are threads that control when a preempted thread gets a new time slice, using an IPC protocol like the page-fault protocol. However, it is an area of ongoing research (and beyond the scope of this thesis) to investigate whether such a preemptor facility is sufficient for implementing versatile user-level scheduling.[1] Consequently, no current L4 kernel implements preemptors.

A task *a*'s threads can create or delete another task *b* (identified by number) only if it owns the right to do so *(task right).* In this case, *a* is said to be *b*'s *chief,* and *b* belongs to *a*'s *clan.* Clans can be nested. Task rights can be passed along between tasks with a system call. When a chief *a* deletes one of its clan members *b*, all tasks in *a*'c

[1] Jean Wolter, personal communication.

clan are deleted as well, as well as their clans, and so on, recursively; task *a* inherits all task rights that were present in *b* and *b*'s subclans.

L4 originally defined a clans-and-chiefs–based security model that restricted direct inter-clan IPC. When a thread *A* sent an IPC to a thread *B* in different clan, the kernel had to intercept the message and send it to the next thread in the chain of chiefs between *A* and *B*. This chief could inspect the message and then pass it along on behalf of *A* using a special mode of IPC, "deceiving IPC." This security model proved to be too inflexible and was abandoned. Fiasco does not place restrictions on inter-clan IPC. It currently has no support for controlling communication.

The kernel interface comprises only seven system calls: synchronous IPC; find nearest IPC partner in a clan; revoke a memory mapping; switch to a different thread; create a thread or modify thread state; modify a thread's scheduling attributes; and create or delete task or pass task right.

L4 kernel implementations do not contain default device drivers or memory managers. Consequently, they cannot load programs from disk. Therefore, such infrastructure must be loaded before booting the kernel.

L4 defines an IPC protocol for the lowest-level memory manager, named σ_0 or *Sigma0* (pronounced "sigma-zero") and defines that Sigma0 can send flexpages for all valid memory and input–output addresses as if it owned mappings for them. Sigma0 is a trusted server that must always be present. However, Sigma0's implementation is not part of the kernel and runs at user level.

In addition to the kernel ABI and the Sigma0 protocol, the specification also requires a number of interfaces for booting, debugging, and profiling.

4.1.2 Design goals

There were three major design goals for Fiasco: real-time properties, speed, and maintainability. In this thesis, I concentrate on real-time and performance issues only; I discuss maintainability in depth in a companion technical report, "The Fiasco kernel: System architecture" [Hoh02].

4.1.2.1 Real-time properties

Fiasco is meant to be used in real-time systems. Such systems have three main requirements: predictability, latency, and flexibility.

Predictability. Real-time systems need to guarantee a worst-case execution time for all operations that are relevant to real-time computing. Such operations include hardware-interrupt handling, scheduling, and certain system calls including IPC.

Low Latency. In addition to bounding worst-case execution times, it is also important to optimize for short reactions to system events such as interrupts and system calls.

Flexibility. There exists a large number of different real-time scheduling strategies. Fiasco must be designed to enable adaptions to a number of scheduling strategies.

Also, the kernel should allow concurrent execution of both real-time and non-real-time applications, and still be able to guarantee real-time properties.

I have addressed predictability by designing Fiasco as a preemptible kernel. That means that interrupts can preempt the kernel virtually anywhere without risking inconsistencies, and can wake up a higher-priority user-level thread in bounded time. The thread with the highest priority can also execute all system calls in bounded time, independent from other threads' states. As the L4 programming interface provides tasks with a very low-level system view, it is possible to build systems that rule out surprises for real-time tasks such as page faults caused by vanishing mappings, or dynamic priority changes.

Preemptability, together with "hard" priorities, also provides the flexibility to run both real-time and non-real-time applications side-by-side on one machine. There is no way for a low-priority, non-real-time thread to prevent a higher-priority thread from accessing a kernel service or resource in bounded time.

As I stated in Section 4.1.1, flexible user-level scheduling on L4 is still an area of active research, and is beyond the scope of this work. Therefore, I did not consider scheduling flexibility as a special real-time requirement—instead, it is more of a maintainability issue, which I discuss in my companion technical report [Hoh02].

I will discuss the low-latency requirement separately in the next section.

4.1.2.2 Speed

The main grief with early microkernels is that these kernels suffered from bad performance. For a long time it was thought that the slowness stemmed from structural problems inherent to microkernel-based systems.

Jochen Liedtke proved this assumption wrong. He laid down design principles that enable performance-oriented kernel design. He also designed the L4 programming interface such that L4 implementations can be fast [Lie95].

I wanted Fiasco to be an efficient L4 implementation. That meant that I needed to follow Liedtke's guidelines when designing kernel internals. However, as maintainability was an important concern, too, I chose to implement in a higher-level language. As a compromise between efficiency and convenience, I selected C++ as Fiasco's implementation language. One of the design challenges was to merge the two aspects of object-oriented modeling with factored, well-isolated components and performance-oriented kernel design.

Speed also is an issue for Fiasco as a real-time system because the worst-case execution time for critical parts of the system should be low. In case of Fiasco, the most important parts are the IPC subsystem and the interrupt-handling paths. Nonblocking synchronization already takes care of not hindering high-priority threads when they become ready to run. However, nonblocking synchronization can have a significant overhead. Therefore, the design methodology I developed and used for Fiasco (see Chapter 3 of this thesis) mandates that kernel-global data shared between noncooperating threads needs to be synchronized using the lowest-overhead scheme possible: lock-free synchronization, and very short critical sections protected with interrupt disabling and—on SMP machines—spin locks.

4.2 Synchronization in the Fiasco microkernel

The Fiasco kernel closely follows the design outlined in Section 3.1. In this section, I report how various data structures are synchronized in this kernel, and I detail the design of my wait-free locking-with-helping mechanism.

4.2.1 Overview of kernel objects

Let me begin by briefly describing the objects the Fiasco microkernel implements. (For a philosophical discussion on what a microkernel should and should not implement, I refer to Liedtke [Lie95].)

For this summary, I have grouped the kernel objects into local state and global state according to the synchronization methodology (Section 3.1.2).

LOCAL STATE.

Threads. The thread descriptors contain the complete context for thread execution: a kernel stack, areas for saving CPU registers, a reference to an address space, thread attributes, IPC state, and infrastructure for locking (more on the latter in Section 4.2.3).

Address spaces. There exists one address space per task. Address spaces implement the x86 CPU's two-level page tables. They also contain the task number, and the number of the task that has the right to delete this address space.

Hardware-interrupt descriptors. Each hardware interrupt can be attached to one user-level handler thread. The kernel sends this thread a message every time the interrupt occurs.

Mapping trees. The Fiasco microkernel, like L4, allows transferring virtual-to-physical page mappings via IPC between tasks. The mapping in the receiving task is dependent on the sender such that when the mapping is flushed in the sender's address space, dependent submappings are recursively flushed as well [Lie95]. Mapping trees are objects to keep track of these dependencies. There is one mapping tree per (physical) page frame.

GLOBAL STATE.

Present list and ready list. These doubly-linked ring lists contain all threads that are currently known to the system, or ready-to-run, respectively. On both lists, the "idle" thread serves as anchor of the list.

Array of address space references. This array is indexed by address-space number. It contains a reference for each existing address space; for nonexisting address spaces, the array contains an address space index referring to the task that has a right to create the address space. The Fiasco microkernel uses this array for create-rights management, and to keep track of and look up created tasks.

Timer and timeouts. The kernel supports timeouts on IPC operations. These timeouts are descriptors that can be registered with a timeout queue that is traversed by the timer-interrupt handler.

Array of interrupt-descriptor references. In this array, the Fiasco microkernel stores assignments between user-level handler threads and hardware interrupts.

Page allocator. This allocator manages the kernel's private pool of page frames.

Mapping-tree allocator. This allocator manages mapping trees. Whenever a mapping is flushed or transferred using IPC, the corresponding mapping tree shrinks or grows. Once certain thresholds are exceeded, a new (larger or smaller) mapping tree needs to be allocated; this behavior is an artifact of the Fiasco microkernel's implementation of mapping trees.

4.2.2 Synchronization of kernel objects

Following my design methodology from Section 3.1.2, the global state should be synchronized using lock-free synchronization, while for local state the overhead of wait-free locking is acceptable. Primarily, I closely adhered to these guidelines. But I also made the requirements somewhat stronger (i. e., I disallowed wait-free locking because of the extra overhead) where performance was critical, and I allowed a small relaxation to gain ease of use where it did not affect real-time properties.

LOCAL STATE. Threads are the most interesting objects that must be synchronized. I accomplish synchronization using wait-free locks (described in Section 4.2.3). However, for IPC-performance reasons I do not lock all of a thread's state. Instead, I defined some parts of thread data to be not under the protection of the lock, and use lock-free synchronization for accessing these parts. In particular, the following data members of thread descriptors are implemented lock-free: the thread's state word, which also contains the ready-to-run flag and all condition flags for waiting for events (as explained in Section 3.3); and the sender queue, a doubly-linked list of other threads that want to send the thread a message. The state word can be synchronized using CAS. For the doubly-linked sender list I use a very short critical section protected by disabling interrupts.[2]

The Fiasco microkernel protects mapping trees, like the bulk of the thread data, using wait-free locks.

Address spaces require very little synchronization. The kernel has to synchronize only when it enters a reference to a new second-level page table into the first-level page table. Deletion does not have to be synchronized because only one thread can carry out this operation: Thread 0 of the corresponding task deletes it when it is itself deleted. Otherwise, I do not synchronize accesses to address spaces: Only a task's threads can access the task's address space, and the result of concurrent updates of a mapping at a virtual address is not defined. As mappings are managed in (concurrency-protected) mapping trees and not in the page tables, mappings cannot get lost, and all possible states after such a concurrent update are consistent.

I did not have to synchronize hardware-interrupt descriptors at all because once they have been assigned using their reference array (global state), only one thread ever accesses them.

GLOBAL STATE. The reference arrays for address spaces and hardware-interrupt descriptors can easily be synchronized using simple CAS.

For the doubly-linked present and ready lists, I use a very short critical section protected by disabling interrupts.[3]

[2]The SMP version uses a spin lock per receiver to synchronize accesses to the sender list.

[3]For the SMP port, this does not present a problem: The ready list is per-CPU, so interrupt-disabling can still be used. Accesses to the present list are rare and can be synchronized using a spin lock.

It is unnecessary to implement the kernel allocators for pages and mapping trees with lock-free synchronization; here I used wait-free locking, as for the local state. I allowed this relaxation of my guidelines in these instances for the following assumption: Threads with real-time requirements never allocate memory (for page tables) or shrink or grow mapping trees once they have initialized. Instead, they make sure that they allocate all memory resources they might need at initialization time. Therefore, real-time threads do not compete for access to these shared resources, and the overhead for accessing them is irrelevant. Should my assertion become false in the future, I will revisit this design decision.

4.2.3 Wait-free locking

The implementation of wait-free locking with helping in the Fiasco microkernel is very similar to the uniprocessor and multiprocessor mechanisms presented in Section 3.3.

The Fiasco microkernel extends the basic wait-free locking mechanism in two respects.

First, locks that protect threads in the Fiasco microkernel (*thread locks*) are furnished with a switch hint. This hint overrides the system's standard policy of scheduling the threads, locking thread or locked thread, once the locker frees the lock. Usually, the runnable thread with the highest priority is scheduled, but the Fiasco microkernel's IPC–system-call semantics dictate that the receiver gets the CPU first.[4] The hint can take one of three values: When the lock is freed, switch to (1) the previously-locked thread, (2) the locker, or (3) to whoever (of the two threads) has the higher priority. To achieve IPC semantics, the sender locks the receiver, wakes it up, and sets the hint to Value 1 before releasing the lock. Value 2 is used when a sender locks a receiver during a long-IPC operation (it is used as an optimization during flexpage transfers in long IPC), and Value 3 is used for all other (non-IPC) thread manipulations.

Second, threads need to maintain a count of objects they have locked. This count is checked in the thread-delete operation to avoid deleting threads that still hold locks.

If one thread is locked by another, it usually cannot be scheduled. If the scheduler or some other thread activates a locked thread, its locker is activated instead. The only exception is an explicit context switch from a thread's locker. The thread-delete operation uses this characteristic to first lock to-be-deleted threads and then push them out of their critical sections. After a thread has released all locks, it can be safely deleted.

The time-slice donation scheme introduced in Section 3.3 requires that nested critical sections do not sleep. During the implementation of the Fiasco microkernel, I did not find this limitation to be very restricting. I completely avoided nesting critical sections that might want to sleep: I found that even for complex IPC operations, there was no need to lock both of two interacting threads.

Instead, a thread *A* that needs to manipulate another thread *B* usually locks *B*, but not itself (*A*). Kernel code running in *A*'s context needs to ensure that locked operations on *A* itself (by a third thread, *C*) cannot change state that is needed during *A*'s locked operation on *B*. In practice, this is very easy to achieve: All locked operations first check whether a change to the locked thread is allowed. If the locked thread is not in the correct state, the locked operation is aborted. All threads explicitly allow a set of locked operations on them by adjusting their state accordingly.

[4]This behavior can be disabled using a flag argument to the IPC system call when strict priority-based scheduling is required in a real-time system.

```
void sleep (unsigned condition)
{
  Thread* thread = current ();

  for (;;)
    {
      unsigned old_state = thread->state;
      if (old_state & condition)
        {
          /* condition occurred */
          break;
        }
      if (CAS (& thread->state,
               old_state,
               old_state & ~Thread_running))
        {
          /* ready flag deleted, sleep */
          schedule ();
        }
      /* try again */
    }

  thread->state &= ~condition;
}

void wakeup (Thread* thread,
             unsigned condition)
{
  for (;;)
    {
      unsigned old_state = thread->state;
      if (CAS (& thread->state,
               old_state,
               old_state | Thread_running
                         | condition))
        {
          /* CAS succeeded */
          break;
        }
    }

  if (thread->prio > current()->prio)
    switch_to (thread);
}
```

Figure 4.1: Pseudocode for the `sleep` and `wakeup` operations. As the condition flag is stored in the same memory word as the scheduler's ready-to-run flag, the `sleep` implementation does not risk a race condition with the `wakeup` code.

In Section 3.3.1 I mentioned the problem of race-free sleep and wakeup when a resource has to be unlocked before sleep can be called. Figure 4.1 shows pseudocode for my sleep and wakeup operations, which I use for thread synchronization in the IPC implementation. As a means to avoid race conditions between sleep and wakeup, I use binary condition flags for synchronization. All condition flags are located in the same memory word that also contains the scheduler's ready-to-run flag. Using CAS, a thread that wants to sleep can make sure that the condition flag is still unset when it removes the ready-to-run flag.

This solution is only applicable inside a kernel, and it restricts the number of condition flags to the number of bits per memory word. For my microkernel, this was not a severe restriction (the Fiasco microkernel needs less than 10 condition flags), but it may become a problem for more complex systems. For such systems, a more general solution (e. g., protecting sleep and wakeup using a simple lock) can be used.

4.2.4 Synchronization of the IPC operation

Inter-process communication, or IPC, is the secure, kernel-assisted message transfer between a sender and a receiver [Lie96a]. In this section, I describe thread synchronization in Fiasco's IPC mechanism. (I describe the IPC mechanism in more detail in my companion technical report, "The Fiasco kernel: System architecture" [Hoh02].)

The IPC mechanism serves as a case study for nonblocking synchronization and is intended to underline two major suppositions I have made in this thesis:

1. A complex protocol that involves concurrent threads and includes thread synchronization and waiting (i. e., voluntary blocking) can be implemented using only nonblocking synchronization primitives.

2. I provide evidence that even for protocols of this complexity, my methodology's restriction of disallowing blocking in critical sections (or, in the monitor terminology of Section 3.3.4, disallowing waiting for events in *nested* monitor methods) is not a severe limitation.

   In fact, I have implemented the protocol without any need to wait for events inside any critical section (nested or nonnested). Therefore, I have not used the monitor implementation of Section 3.3.4; instead, waiting occurs only outside critical sections.

OVERVIEW. From a user's point of view, L4 IPC is synchronous, that is, senders block until a thread receives their message. For the kernel that means that synchronization between threads involved in an IPC is necessary.

Fiasco uses the wait-free synchronization mechanism described in Section 4.2.3 to synchronize with, that is, *lock* other threads and to carry out complex operations on them. However, as critical sections are not allowed to block, they must not access the user part of the virtual address space. It follows that in order to transfer memory-buffer–based messages (i. e., long IPC), the IPC operation must be split into several nonblocking critical sections that implement a stateful protocol between threads doing an IPC handshake, and potentially blocking memory accesses must be done outside critical sections.

In other words, both sender and receiver pass through a number of IPC states, constituting four basic IPC phases. I first discuss the memory representation of IPC

State	**State flags used in IPC**								
	ready	recv/ wait	ipc	send	busy	busy long	poll	poll long	cancel
Sender states									
send prepared	+		+	+			+	–	
sleeping	–		+	+			+	–	
woken up	+		+	+			–	–	
long IPC in progress	+		+	+			–	–	
page fault in IPC window	+		+	+			–	+	
page-in wait	–		+	+			–	+	
after send	+			–					
Receiver states									
setup	+	+							
prepared	+	+	+	–	–	–			
going to rendezvous	+	+	+	–	+	–			
waiting	–	+	+	–	+	–			
in long IPC	–	+	+	–	–	+			
page-in	+	+	+	–	–	+			
after receive	+		–						
Error conditions									
Timeout	+		–						
IPC canceled	+		–						+

Legend: + = flag set; – = flag cleared; otherwise, flag can be set or cleared.

Table 4.1: Sender and receiver states, expressed in state flags.

states in Subsection 4.2.4.1, before I discuss IPC phases and walk through their IPC states in Subsection 4.2.4.2.

The sender uses locked operations to asynchronously modify the receiver's state, and vice versa. Two locked operations cannot rely on thread states remaining unmodified in-between because other operations (like an *lthread_ex_regs* system call) can terminate the IPC operation for one partner. Therefore they always verify that the other thread is in the correct state, and abort the IPC if that is not the case. I discuss error conditions and guarding against them in Subsection 4.2.4.3.

In this thesis, I do not discuss how Fiasco translates hardware interrupts and page faults to IPC messages. Please refer to the companion technical report for information on that subject [Hoh02].

4.2.4.1 IPC states

As I already mentioned, Fiasco implements IPC as a stateful protocol between a sender and a receiver. Figures 4.2 and 4.3 show an overview of sender and receiver states.

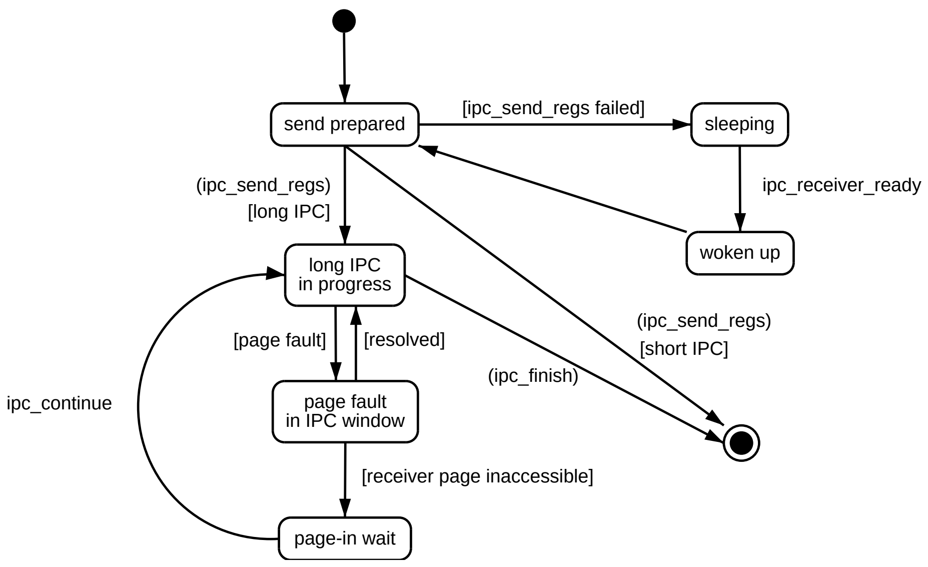


Figure 4.2: Sender state chart.

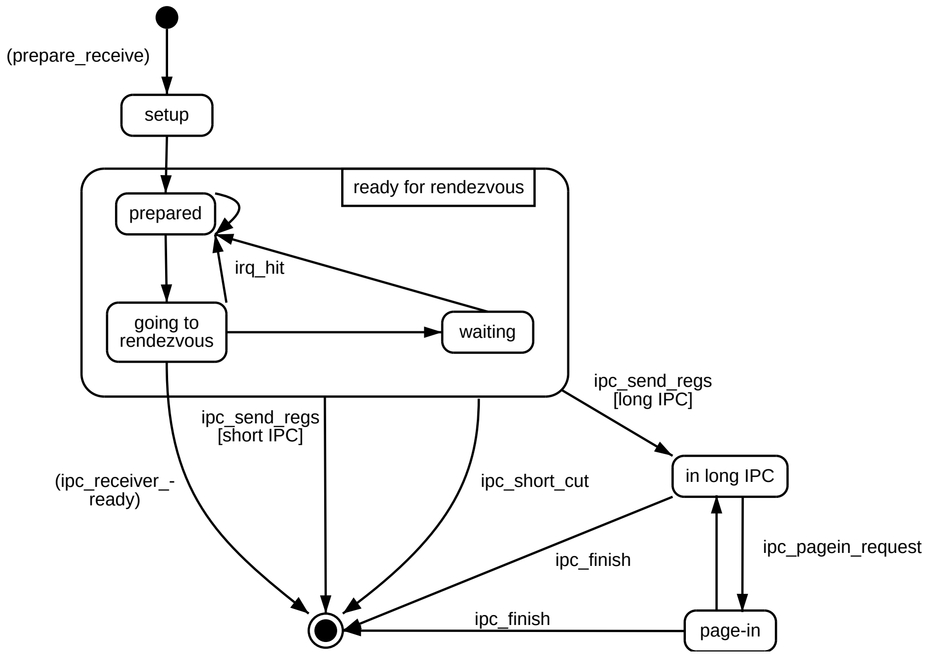


Figure 4.3: Receiver state chart.

As senders and receivers not only keep track of their own state but also modify their IPC partner's state as part of the IPC protocol, the IPC states need an explicit data representation. Fiasco keeps IPC states as part of each thread's state word. It also uses a number of flags in the state word to avoid race conditions when sender or receiver go to sleep. Table 4.1 defines how IPC states translate into state flags.

As is obvious from Table 4.1, IPC states are defined by a bit pattern in the state word. Each state transition changes the bit pattern in a unique way (but two states can have the same pattern if there is no transition between them). Not all bits are significant for each state. This feature allows a thread to stay in receiver–setup state while carrying out a send operation. Therefore, for a combined send–receive IPC system call, Fiasco can set up the receive operation before carrying out the send operation, allowing it to atomically switch from a sender state to a receiver state by clearing the "send" flag. In the sender states, the additional flags used for receiving are irrelevant.

State transitions must be guarded, that is, Fiasco needs to check whether an IPC state it modifies is still valid. As Fiasco allows preemption and parallel execution of sender and receiver, state checking and modification need to be atomic from the IPC partners' point of view. In other words, Fiasco must synchronize accesses to and modifications of IPC states to ensure atomic state transitions.

Fiasco synchronizes simple state transitions using lock-free compare-and-swap–based atomic update. More complex transitions such as the IPC handshake between sender and receiver are implemented with critical sections locked using Fiasco's wait-free synchronization mechanism (Section 4.2.3). Section 4.2.4.3 covers IPC-state synchronization in more detail.

4.2.4.2 IPC walk-through

Both the send and the receive IPC operations are logically structured into four phases: setup, rendezvous, data transfer, and finish.

SETUP AND RENDEZVOUS. During an IPC system call with both send and receive parts, the send operation always takes place before the receive operation. However, the receive setup phase actually executes before the send operation's phases to facilitate atomic switching from the send operation to the receive operation—a behavior the L4 specification requires: Threads must accept reply IPC immediately (without requiring a timeout in the sender).

During rendezvous, the sender takes the active role while the receiver remains passive. The receiver basically sleeps through the whole procedure; the sender only wakes up the receiver if the receiver needs to page in a virtual-memory page (in the receiver's address space), or when the IPC has been finished.

Figure 4.3 shows a state diagram for the receiver. To enable IPC rendezvous, the receiver first puts itself into state "setup," followed by state "prepared." The two states are separate to make it possible to carry out a send operation between the two and atomically switch from the send operation to the receive operation's "prepared" state.

Once the receiver has entered the "prepared" state, senders can asynchronously rendezvous using method *ipc_send_regs*, for example, by putting the receiver into the final state (for short IPC) or into state "in long IPC" (for long IPC).

However, the receiver would normally proceed to state "going to rendezvous" where it checks the if a sender has queued in the receiver's sender queue. If that is the case, the receiver wakes up the sender using the *ipc_receiver_ready* operation; the sender becomes active, rendezvouses, and executes the IPC (putting the receiver into the final state or state "in long IPC").

If there is no sender waiting for a rendezvous, the receiver tries to proceed to state "waiting" where it sleeps until a sender rendezvouses. The transition only succeeds if no sender has already asynchronously put the receiver into another state ("prepared," "in long IPC," or final), in which case execution proceeds there.

Meanwhile, the sender prepares for IPC by entering state "send prepared" (see Figure 4.2). It queues in the receiver's sender queue and attempts a first rendezvous using *ipc_send_regs*. If successful, the sender directly proceeds to its final state (for short IPC) or "long IPC in progress" (for long IPC). Otherwise, it starts sleeping by going to state "sleeping." When a receiver sends a *ipc_receiver_ready* request, the sender continues in state "woken up." It then switches back to state "send prepared," where it retries the *ipc_send_regs* operation.

DATA TRANSFER. In receiver state "in long IPC," a sender has rendezvoused and started a long IPC. The receiver continues to sleep, waiting either for the IPC to finish, or for the sender to request a page-in using *ipc_pagein_request* (which puts it into state "page-in"). In the latter case, the receiver tries to page in user data (using flexpage IPC), and subsequently resumes the sender using *ipc_continue*.

After the initial rendezvous for a long IPC, the sender continues in state "long IPC in progress." It maps the receiver's message buffer and indirect-string buffer(s) into two regions of the kernel's virtual address space, the *IPC windows*. These mappings allows the sender to copy data using a simple memory-copying routine. During memory-copying, page faults in the sender's address space are handled by the usual user-paging mechanism. Page faults in the IPC windows put the sender into state "page fault in IPC window," where it sends an *ipc_pagein_request* to the receiver; then, the sender goes to sleep (state "page-in wait") and waits for the receiver's *ipc_continue* to resume it in state "long IPC in progress."

Senders in state "long IPC in progress" also transfer flexpage mappings using *ipc_send_fpage*.

The sender finishes off its send operation using *ipc_finish* (or a variant of *ipc_send_fpage* if just one flexpage is sent using short IPC) and goes to its final state.

FINISH. For the receiver, IPC finishes when the "ipc" state flag is removed from its state word (refer to Table 4.1). This happens when one of *ipc_send_regs*, *ipc_send_fpage*, and *ipc_finish* puts it into its final state, or when an error condition (timeout or cancel; see next subsection) removes that flag. Whenever it awakes, the receiver has to check for this condition.

The send operation also finishes when an error condition removes the "ipc" state flag, and it also has to monitor its state word to detect this condition. Otherwise, the send operation ends when the "send" flag is removed, that is, when the sender enters its final state. When reaching that state, the sender needs to check if the send operation is followed by a receive operation. If not, the "ipc" state flag is removed also.

4.2.4.3 Asynchronous state changes

During IPC, a thread's state can change asynchronously because of any of the following three reasons: An IPC error occurred, another thread executed a locked operation on the first thread, or an interrupt IPC is being delivered. I discuss the first two cases in this section; please find information on interrupt IPC in my companion technical report, “The Fiasco kernel: System architecture” [Hoh02].

There are three error conditions that can lead to abortion of an ongoing IPC operation:

Timeout. When an IPC timeout has been specified and occurs, a timeout handler will execute. This handler will modify the thread’s state: It removes the “ipc” flag, and it adds the “ready” flag.

This change can occur at any time, even during the execution of locked operations. Therefore, “normal” kernel code related to IPC has to guard against this error condition, and locked operations at least need to be aware of it.

Cancel. When a thread executes a *lthread_ex_regs* system call on another thread involved in an IPC, it runs an *initialize* method on that thread. The *initialize* locked operation removes the “ipc” flag and adds the “ready” and “cancel” flags.

As this modification is implemented in a locked operation, it cannot affect other locked operations such as *ipc_send_regs*, but it can occur during execution of normal kernel code. Therefore, normal code has to guard against this error condition; locked operations need to check the thread’s state only once at the beginning.

Kill. When a thread is killed using a *kill* method, it will have all of its state removed, and its state word will be cleared out, terminating ongoing IPC.

Again, as this modification is implemented in a locked operation, it cannot affect other locked operations, but it can occur during execution of “normal” kernel code. However, once a thread has been killed, its kernel code will stop being executed. Therefore, it cannot directly guard against this error condition. Instead, the thread needs to record any resources that depend on it so that the *kill* method can free them.

Guarding against the first two error conditions works as follows: As the thread’s state can be asynchronously modified, state transitions in normal kernel code must be done atomically. When changing the state using the atomic `CMPXCHG` instruction, it is possible to detect beforehand if one of the error condition has occurred. If this is the case, the thread must not go to sleep again; instead, it must return to the user immediately.

If the IPC actually finishes (without further blocking) even though a timeout or a cancel occurred, the user does not have to be made aware of the error. Otherwise, the IPC must be aborted and the error must be signaled. This may include waking up the IPC partner (and signaling it the error as well) if it depends on the thread that has detected the error.

In locked operations, it is sufficient to guard the whole operation with a state check at the beginning. This check ensures that the thread the operation works on still is in the required state. For example, *ipc_finish* needs to verify that the receiver is state “in long IPC” or “page-in” (Figure 4.3); otherwise, some error condition has interfered with the ongoing IPC, and the operation needs to be aborted.

Once locked operations have done the initial check, they can carry out their work without monitoring the state word. They cannot be superposed with other modifications on the thread (including the thread’s state word) except timeouts; however, these operations do not have to care about timeouts because they themselves never block

Operation	**Precondition**	**State modification**	**Action**
irqhit	ipc && recv/wait	+ready, –busy	queue interrupt sender
timeout	ipc	+ready, –ipc	timeout
initialize	none	+ready, +cancel, –ipc	reset thread
kill	none	deletes all flags	delete thread

Table 4.2: State modifications some asynchronous operations carry out during an IPC

(they are not allowed to). They just have to make sure that state modifications made by the timeout handler are not accidently undone. Therefore, locked operations also use the atomic `CMPXCHG` instruction to modify state words.

The bottom of Table 4.1 shows IPC-state flags modified by error conditions against which Fiasco's IPC code must guard. Table 4.2 shows the error conditions in context with other asynchronous operations the IPC mechanism must handle.

Chapter 5

Performance evaluation

In previous chapters, I have developed a methodology for building real-time systems using nonblocking synchronization, and I have described the Fiasco microkernel, an operating-system kernel I developed using my methodology. In this chapter I analyze the performance of the synchronization primitives my approach proposes. Specifically, I look at the implementations of these primitives in Fiasco.

I present two different kinds of performance results. In Section 5.1, I look at minimal performance overheads imposed by the synchronization mechanisms, for the uniprocessor and the multiprocessor version of Fiasco. Section 5.2 presents worst-case real-time performance results; in particular, it gives a measure of the interruptibility I achieved for Fiasco by determining how long Fiasco disables interrupts in the worst case.

5.1 Microbenchmarks

In this section, I assess the practicability of nonblocking synchronization on x86 CPUs. I look at best-case performance overheads for both lock-free synchronization and the uniprocessor and multiprocessor helping mechanisms I introduced in Sections 3.3.1 and 3.3.2, in their implementation in the Fiasco microkernel, and I compare these costs with those of alternative implementations of critical sections.

In another measurement, I compare the cost of my current wait-free lock implementation with that of my previous Massalin–Pu serializer-style synchronization mechanism (described in Section 3.3.5).

5.1.1 Measurements and results

Table 5.1 summarizes my findings about synchronization overheads. The upper part of the table shows the cost of uniprocessor implementations of the synchronization mechanisms, and the lower part shows the multiprocessor results. I have included results for mechanisms implementing both lock-free synchronization—atomic memory update using interrupt-flag manipulation and CAS—and wait-free synchronization—Fiasco's helping-lock mechanism.

ATOMIC MEMORY UPDATE. I measured the overhead of updating one or two memory words using an atomic counter, CAS, and CASW. On uniprocessors, any memory

Synchronization operation	**Duration**					
	Pentium III, 450 MHz			Pentium 4, 1.6 GHz		
	[cycles]	[ns]	error	[cycles]	[ns]	error
Uniprocessor operations:						
atomic counter	7	16		3	2	
CAS	11	24		18	11	
CASW	19	42		166	104	
Interrupts off+on	23	51		78	49	
lock without contention	198	440	−4%	216	135	−2%
lock with helping	760	1689	−8%	1296	810	−12%
context switch	172	382		216	135	
Multiprocessor operations:						
atomic counter (bus-locked)	34	76		140	88	
CAS (bus-locked)	35	78		147	92	
CASW (bus-locked)	40	89		213	133	
lock without contention	294	653	−2%	668	418	−1%
lock with helping	1668	3707	−4%	3830	2394	−4%
context switch	381	847		984	615	
raw IPI latency	1149	2553		1500 ?	938 ?	

Table 5.1: Duration of synchronization operations. I evaluated the helping-lock implementation of the Fiasco microkernel; for comparison, I provide the cost of simple synchronization primitives. All values are best-case results; they allow assessing a minimum overhead and cannot be used for worst-case execution-time analysis.
The "error" columns show the maximal measurement overhead of the given measurement. The question marks "?" indicate that I could not measure IPIs on the Pentium 4 because I do not have access to a multiprocessor machine with Pentium-4 CPUs; however, I made an educated guess (explained in the text).

Synchronization operation	**Duration**			
	Pentium, 133 MHz		Pentium II, 400 MHz	
	[cycles]	[µs]	[cycles]	[µs]
current Fiasco thread lock	245	1.842	245	0.613
old Massalin–Pu–style thread lock (includes two context switches)	627	4.714	607	1.518

Table 5.2: Comparison of Fiasco's current thread lock with previous synchronization mechanism based on Massalin–Pu-style serializer.

update can be made atomic by turning off interrupts (and preventing page faults) for the duration of the critical section, so I have included the overhead induced by this mechanism as well.

The multiprocessor versions of these primitives often are significantly more costly than the uniprocessor versions, especially on the Pentium 4. This is so because in the multiprocessor version, these primitives require memory-bus lock ("`lock`" prefix in x86 assembly language), which costs between 21 and 27 additional cycles on the Pentium III and about 130 cycles on the Pentium 4.

HELPING. I measured the cost of my helping mechanism (uniprocessor and multiprocessor variants) on a no-operation critical section for both the uncontented and the contented case. To simulate contention, I set up a (low-priority) thread that acquired the lock and then released the CPU; I then measured the elapsed time in another thread from entering a critical section, helping, and leaving the critical section. Helping includes two context switches (to and from the context of the helped thread, which releases the lock immediately when activated).

I have included a "measurement error" percentage in Table 5.1 because the measurement overhead was slightly different in the contented and uncontented experiments. In the uncontented case, it is that of a loop which runs the experiment 10000 times. In the contented case, I had to factor out the cost to set up the experiment (initialization of low-priority thread), which required two extra measurements in each loop cycle. As the cost of the "read time-stamp counter" instructions I used varies depending on the CPU's pipeline state, I cannot simply subtract a fixed cost from the measured elapsed times. Instead, I present times including measurement overhead, and maximum error values. These uncertainties are small enough to still allow meaningful comparisons.

In the multiprocessor version, locking an object when there is no contention costs 294 and 668 cycles (on Pentium III and Pentium 4, respectively), whereas helping costs an additional 470 percent (on both CPUs).

Multiprocessor versus uniprocessor versions. The table indicates that a helping-lock implementation is significantly more costly on multiprocessors than on uniprocessors: On the Pentium III, the uncontented case has an additional cost of 48 percent, and locking with helping adds 119 percent. On the Pentium 4, the values are even worse: 209 percent and 196 percent. Apparently the main source of extra overhead are the context-switch operations.

I tracked down this extra cost to my implementation's use of atomic memory-update instructions such as CAS. As I stated in a preceding paragraph, these instructions are relatively costly, especially on the Pentium 4.

IPI. I also measured the minimum IPI latency. In this test, one thread interrupts execution on another CPU using an IPI, and the interrupt handler on the other CPU signals arrival of the IPI by changing one memory word. The latency is the elapsed time on the first CPU from sending the IPI to noticing the change in the memory word. It does not include a context switch on either CPU.

I could carry out this measurement only on the Pentium III, where I determined a latency of 1149 cycles or 2553 nanoseconds. On the Pentium 4, I expect a latency of about 1500 cycles: The IPI transmission is likely faster on the Pentium 4 (because on this CPU, IPIs are transferred on the processor bus, not on the APIC bus [Int99]), but interrupts need more cycles than on the Pentium III. For example, the Pentium 4 needs 1168 cycles to enter the kernel when the CPU's local APIC generates an interrupt (Pentium III: 203 cycles).

SINGLE-SERVER PERFORMANCE. Table 5.2 compares Fiasco's current thread lock with my previous synchronization mechanism based on a Massalin–Pu-style serializer.

The current lock implementation evaluated here is mostly equivalent to the uniprocessor helping lock of Table 5.1 (line "uniprocessor lock without contention"). However, the overhead I measured for this lock is slightly different in Table 5.2 because of two reasons: First, I chose to compare the Massalin–Pu mechanism against a thread lock, not a plain lock, because my Massalin–Pu implementation by its nature always locks a thread (see Section 3.3.5); thread locks differ from plain locks in that they are extended with a switch hint (see Section 4.2.3). Second, the measurement was conducted on somewhat older hardware.

This measurement indicates that the performance of the Massalin–Pu solution is limited by the context-switch overhead. The new solution is more than twice as fast than the old one.

5.1.2 Discussion

COST AND BENEFIT OF NONBLOCKING SYNCHRONIZATION. Obviously, nonblocking synchronization comes with a cost. The cost can be as low as 3 cycles (atomic counter on uniprocessor Pentium 4) and as high as some thousand cycles (3830 cycles for helping lock under contention on multiprocessor Pentium 4).

However, there are also benefits to nonblocking synchronization: When compared to protecting (long) critical sections by disabling interrupts, the system remains interruptible during critical sections, providing lower interrupt latency and better scheduling accuracy. Nonblocking synchronization also prevents priority inversion, which ultimately leads to lower worst-case execution times. Additionally, lock-free synchronization removes lock contention from multiprocessor system, and Michael and Scott have shown that lock-free implementations scale better than lock-based approaches on multiprocessors [MS96].

Disabling interrupts as a lock-free protection mechanism. In the preceding paragraph I assumed that interrupt-disabling is used only to protect very short critical sections (such as a system-level CAS2 implementation). As long as these critical sections take less cycles than the longest possible atomic CPU operation (e. g., a system call using the `int` instruction on the Pentium 4 takes 796 cycles), interruptibility is not adversely affected.

When this condition is true for *all* critical sections of an object, *and* interrupt-disabling can be used to protect this object (i. e., on uniprocessors, or for CPU-local data on multiprocessors), this lock-free protection mechanism can be less costly than using the wait-free helping lock.

In Fiasco, this condition clearly was not true for many critical sections, and some critical sections cannot be made shorter (i. e., preemption points cannot be introduced) without restructuring the kernel. However, the future L4 interface (dubbed L4 version 4) has been carefully engineered to not require long critical sections in the kernel, so this technique may become a viable alternative for implementations of that interface.

Helping versus IPI on multiprocessors. An alternative to cross-CPU helping (i. e., executing a blocked thread's critical sections on remote CPUs) would be to use strictly CPU-local resources. To access such a resource from a different CPU, remote threads would have to run their critical sections by invoking a cross-CPU RPC implemented using IPIs. In this scenario (which I call "IPI solution"), accesses to remote resources *always* require an IPI, independent from contention.

From Table 5.1 I can infer that entering a critical section using an IPI needs significantly more elapsed time in the uncontented case, whereas the IPI solution can have an advantage when the critical section is blocked. The actual blocking time depends on the nature of synchronization required on the remote CPU before the critical section is started. For example, there can be overhead for scheduling the critical section, or—if critical sections are protected by enabling and disabling interrupts—the IPI may be blocked before it is actually handled.

After the critical section has been completed, the remote CPU needs to notify the first CPU, which can be realized synchronously (first CPU polls) or asynchronously (remote CPU replies with an IPI). In both cases, the total number of cycles spent on both CPUs to handle the RPC likely is higher than with helping (even in the contented case).

Uniprocessor synchronization for global resources. Compared to the uniprocessor synchronization primitives, the multiprocessor-safe operations are relatively expensive. This effect is especially serious on the Pentium 4, and I expect it to occur on all modern CPUs with high internal clock frequencies and long pipelines.

When remote resource accesses are infrequent, it is possible to reduce synchronization overhead by using an uniprocessor synchronization scheme and carrying out remote accesses with RPCs (as I discussed in previous paragraphs) or by transparently falling back to a multiprocessor synchronization scheme (e. g., using a form of specialization [PAB+95]).

Using pairs of enabling and disabling interrupts is especially attractive as a local synchronization scheme because of these operations' low cost. This method creates critical sections that are not preemptible, which can lead to a higher blocking time. Therefore, interrupt disabling is best suited for very short critical sections where the total time of the critical section is only a fraction of other methods' pure overhead.

For longer critical sections where preemptability is a concern, I suggest my uniprocessor helping lock.

Expensive atomic operations and possible optimization. The high cost of multiprocessor-safe atomic operations on the Pentium 4 was quite surprising, given that this CPU does not actually lock the memory bus but synchronizes atomic memory accesses using its cache-coherency protocol.[1] Apparently the Pentium-4 designers did not find these operations interesting enough for inclusion in the CPU's high-performance RISC core. Also, cache synchronization costs more CPU cycles on the Pentium 4 because of the larger difference between CPU and cache-memory clock rates.

I have carried out a study of the cost of various atomic memory operations on the Pentium III and Pentium 4 CPUs, to gain hints for a future optimization of my locking primitives. Removing the need for the `cmpxchg8b` instruction (which atomically compares and exchanges 8 contiguous bytes in memory) seems to be especially rewarding

[1] I verified this behavior by analyzing the CPU's performance counters.

as this instruction alone is more than five times as expensive as on the Pentium III. Optimizing my mechanism using this data still remains an area of future work, though.

SERIALIZER VERSUS WAIT-FREE LOCK. My wait-free lock implementation has significantly lower overhead than the Massalin–Pu serializer. The reason is that in the uncontented case, thread switches become unnecessary. Besides the performance improvement, the lock solution is also significantly easier to use because it directly supports the mutual-exclusion programming paradigm.

5.2 Real-time characteristics

To assess Fiasco's real-time performance, I ran a minimal interrupt-latency benchmark under worst-case conditions.

In this experiment, I compared Fiasco with RTLinux, a Linux kernel extended with a small real-time executive [YB97]. RTLinux is known to have excellent real-time–scheduling properties [Meh99], which justifies using it as a comparison metric for Fiasco.

For an accurate and fair comparison, on top of Fiasco I used a combination of software that provides the RTLinux API. This allowed me to run the same benchmark on both systems.[2] In particular, this software, taken from the DROPS system [HBB$^+$98], consisted of the following components:

- L4RTL, an implementation of the RTLinux API on top of the L4 interface [MHSH01].
- L^{4}Linux, a Linux server for time-sharing programs that runs as an application program on top of Fiasco and is binary-compatible with original Linux [HHL$^+$97]. L^{4}Linux was configured to use locks instead of interrupts for internal synchronization to prevent its device drivers from disabling interrupts and thereby inducing scheduling delays [HHW98]. In other words, L^{4}Linux can be preempted by real-time tasks at any time.

The main architectural difference between RTLinux and the L4RTL–DROPS combination is that L4RTL real-time threads run in their own user address spaces while in RTLinux, all real-time threads run within the Linux kernel in kernel mode.

In this experiment, I measured the time between the occurrence of a hardware event that triggers an interrupt and the reaction in an RTLinux real-time thread. I conducted the experiment for both original RTLinux and L4RTL.

5.2.1 Experimental setup

To induce worst-case system behavior, I have used two strategies.

First, prior to triggering the hardware event, I configured the system such that the kernel's and the real-time thread's cache and TLB working sets they need to react to the event are completely swapped out (and the corresponding 1st-level and 2nd-level cache lines are dirty).

[2] I used a PC with an 800-MHz Pentium III Coppermine CPU, 256 MByte RAM (100 MHz SDRAM) and 256 KByte 2nd-level cache. Both RTLinux and L4RTL were based on Linux kernel version 2.2.18. I used version 3.0 of RTLinux.

Second, I exercised various code paths in RTLinux, L^{4}Linux, and the Fiasco microkernel. These coverage tests are a probabilistic way to reveal code paths with maximal execution time while interrupts are disabled. Additionally, they increase confidence that the DROPS components and the Fiasco microkernel indeed are completely preemptible. To avoid missing critical code paths because of pathologic timer synchronization, I varied the time between triggering two interrupts.

For cache- and TLB-polluting purposes I used a Linux program that invalidates the caches. It ensures that all 1st-level and 2nd-level cache lines are dirty and need to be written back to main memory when the hardware event occurs. Note that all of the first-level and the second-level cache lines mapping to a specific memory address can contain dirty data from different memory addresses. Therefore, in the worst case a single memory-read operation results in three memory accesses: write-back of a dirty first-level cache line, write-back of a dirty second-level cache line, and finally the read that was actually intended. This case, also known as the *double-purge case*, can occur only in systems that have multi-associative 1st-level caches and that do not provide the *inclusion property* [LH01]. The inclusion property states that the contents of the 1st-level cache also have to be in the 2nd-level cache. Usually this property is needed to guarantee cache consistency on multiprocessor systems where only the highest-level caches take part in a cache-consistency protocol such as MESI. However, on x86 CPUs both the 1st-level and 2nd-level caches independently take part in the MESI protocol, making inclusion unnecessary [Int99]. As a result, inclusion is not required on x86 CPUs.

Because the cache-flooding program uses more memory pages than TLB entries, it has the side effect of flushing the TLB.

For code coverage I used a benchmarking suite for UNIX, hbench [BS97]. This benchmark is designed to provide excellent coverage for Unix systems such as Linux and L^{4}Linux, and I have verified using a profiling tool that it also provides wide coverage for the Fiasco microkernel.

As a measurable and reliable interrupt source, I have used the x86 CPU's built-in interrupt controller (Local APIC[3]). This unit is measurable as it offers a timer interrupt that can be used to obtain the exact time between the hardware event and the reaction in kernel or user code. When the Local APIC's timer interrupt is used in periodic mode, its overhead is close to zero because first, it does not require repeated reinitialization, and second, the elapsed time since the hardware trigger can be read directly from a hardware register. The Local APIC is reliable because the chance of a programming error accidently blocking it or preventing its reactivation is very low.

The drawback of this interrupt source is that unlike other interrupt sources, it cannot be given a higher hardware priority than other interrupt sources. In other words, except for disabling *all* interrupts in the CPU, it is impossible to globally specify which other interrupts must not occur until this interrupt has been acknowledged. I have therefore simulated hardware-interrupt priorities by manually disabling interrupts in the external PIC (*not* in the CPU) until the user-level interrupt handler in the L4RTL measurement thread has acknowledged its interrupt.

With this precaution in place, my interrupt source has the system's highest priority. It cannot be blocked by any other interrupts, and no other interrupt can preempt my interrupt's handler—not even the system's timer interrupt.

[3]APIC = advanced programmable interrupt controller

5.2.2 Measurements

5.2.2.1 What I measured

For both RTLinux and L4RTL, I measured the time between the occurrence of the hardware event and the first instruction in the real-time thread. I measured this time under four conditions: Average case (no additional system load), under hbench, under cache-flooding, and under a combination of the hbench and cache-flooding loads.

Additionally, I measured the time between the occurrence of the hardware event and the first instruction of the kernel-level interrupt handler in both RTLinux and the Fiasco microkernel. This measurement quantifies the effect of critical code sections that disable interrupts within these kernels.[4] By measuring both kernel-level and real-time–thread latencies, I was able to separate overhead induced by kernel code that executes with interrupts turned off from overhead caused by the kernels' implementations.

5.2.2.2 Expectations

In this subsection I estimate the worst-case cost for invoking the user-level interrupt handler for both RTLinux and Fiasco. First I assess the total cost to invoke the handler including address-space switching cost and kernel-entry cost, but assuming the kernel is ideally preemptible. Then I estimate the worst-case time during which interrupts are disabled in the kernels. These two figures are equivalent to the two separate overheads measured in this experiment, except that the initial kernel entry (800 cycles = 1 µs) is accounted for in my user-level–handler–invocation estimation, not in the interrupt-blocking–time estimation.

ENTERING A USER-LEVEL HANDLER. In the following I estimate the activation cost of an interrupt handler in an RTLinux application for both RTLinux and L4RTL under worst-case conditions.

A trap to enter kernel mode induced by a hardware interrupt universally takes about 1 µs—that is, about 800 cycles on my test machine. I measured a memory-read operation that does not hit the cache to cost around 98 cycles. A TLB miss that cannot be satisfied from the second-level cache costs twice as much—around 198 cycles.

When the CPU needs to write data to main memory, it uses a special cache called "write buffer." This cache stores the written data and asynchronously writes them out to main memory when the memory bus becomes available. As long as a write buffer can be allocated in the CPU, the memory-access cost is not worsened by dirty cache lines. I expect that the number of available write buffers (four cache-line–sized buffers, i. e., 4×32 bytes) is sufficient to temporarily cache write-back data from dirty cache lines. I set up an extra experiment to verify this expectation.

For RTLinux, I estimated 8 TLB misses (2 instruction-TLB misses and 6 data-TLB misses) and about 3000 CPU cycles. I guessed the number of cache-line misses to be about 250—equivalent to touching about 8 KByte worth of code and data.

For the L4RTL system, I have to distinguish between kernel code (Fiasco's trap handler and interrupt-to-IPC translation) and user code (L4RTL's handler). In the Fiasco microkernel, I believe the code path to be somewhat longer than in RTLinux (4000 cycles), but the data set to be more local (250 cache-line misses despite the larger number of code cycles). For the L4RTL code running in user mode, I estimate another 8 TLB misses (2i/6d), 10 cache-line misses, and 300 CPU cycles.

[4] L^4Linux is not an issue here, because it never disables interrupts for synchronization.

Event	**Number**	**Cycles / Time**
Kernel trap + hardware	1	800
TLB misses	8	1548
Cache misses	250	24500
Kernel module code		3000
Total Sum		29884 (37.4 µs)

Table 5.3: List of events and estimated costs responsible for delay to activate application interrupt handler in RTLinux

Event	**Number**	**Cycles / Time**
Kernel trap + hardware	1	800
TLB misses	8	1584
Cache misses	250	24500
Kernel code		4000
Kernel mode sum		30884 (38.6 µs)
Return to user	1	70
Cache misses	10	980
TLB misses	8	1584
User code		300
User mode sum		2864 (3.6 µs)
Total Sum		33818 (42.3 µs)

Table 5.4: List of events and estimated costs responsible for delay to activate application interrupt handler in Fiasco and L4RTL

Tables 5.3 and 5.4 summarize my estimations for RTLinux and L4RTL, respectively.

The 37.4 µs I expect for RTLinux are higher than the 15 µs claimed in [RTL01]. This can either be the result of a too pessimistic approach on my part or an underestimation of the worst-case cache-miss costs by the authors of RTLinux.

INTERRUPT-BLOCKING TIME. In an RTLinux system, the only component that disables interrupts is the RTLinux real-time executive itself. Like L^{4}Linux, RTLinux uses locks for low-level synchronization. RTLinux enables and disables only “soft interrupts,” not hardware interrupts. In other words, RTLinux can be blocked only by itself.

Internally, RTLinux does disable interrupts for synchronization. The by far longest and most costly code path during which interrupts are disabled is the interrupt-delivery path—the code path I examined in Table 5.3. Therefore, I assume that RTLinux’ worst-case interrupt-blocking time is close to the 37.4 µs I estimated in that table.

If my assumptions are true that (1) the same RTLinux code path is responsible for both worst-case interrupt-blocking time and handler-invocation time and (2) this code path is considerably more costly than other code paths, I should expect to find that the total worst-case time is considerable less than the sum of the two constituent worst-case times because hitting the worst case for the first part means that cache has been preloaded with code and data for the second part.

Event	Number	Cycles / Time
TLB misses	8	1548
Cache misses	50	4900
Timer-chip acknowledgment	1	10000
Kernel code		1000
Total Sum		17448 (21.8 µs)

Table 5.5: List of events and estimated costs in Fiasco's timer-interrupt handler

For Fiasco, I know from profiling that the timer-interrupt handler is the longest code path in which interrupts are disabled. The timer-interrupt code that runs with disabled interrupts needs to acknowledge the system board's clock chip (which is very costly because of Fiasco's conservative two-stage acknowledgment routine that includes 25 `in` and `out` instructions!) and processes the timeout queue (which contains up to one element in this benchmark—the L^{4}Linux' timer thread's timeout). I guess that this code takes about 1000 CPU cycles, 8 TLB misses (2i/6d) and about 50 cache-line misses. I summarize my estimation in Table 5.5.

As this worst-case interrupt-blocking code path is not equivalent to the code path that invokes the user-level interrupt handler, I expect that the worst cases of these two code paths add up in the total worst-case time (21.8 µs + 42.3 µs = ca. 64.1 µs).

In RTLinux, an important consequence of disabling interrupts in the handler-invocation path is that another incoming hardware interrupt cannot break into the execution of this path. Fiasco, on the other hand, enables interrupts in this path, allowing an interrupt with a higher hardware priority to intercept[5] and cause a higher worst-case execution time. In this experiment, however, I arranged that the measured interrupt always has the highest priority (see Section 5.2.1).

5.2.2.3 Results and discussion

The diagrams in Figures 5.1 and 5.2 show the densities of the interrupt-response times under the four load conditions: no load, hbench, cache flooder, and a combination of hbench and cache flooder.

FIASCO/L4RTL RESULTS. The measured worst-case time for L4RTL was 85 µs. The maximal time to invoke the Fiasco microkernel's handler was 53 µs, and the maximal time to start the corresponding L4RTL thread was 39 µs (Figure 5.2, bottom row).

As the sum of these separate maximums is close to the total maximum, it seems that the code path that disables interrupts and the code path that starts the L4RTL thread indeed use different code and data and are accounted for separately as I predicted in the previous subsection.

RTLINUX RESULTS. The worst-case latency for RTLinux was 68 µs.

RTLinux obviously has long periods of disabled interrupts, too. It takes up to 53 µs to invoke a kernel handler. The fact that this value is close to the total maximum of 68 µs suggests that the code that keeps interrupts disabled is close or identical to

[5]By "intercept" I mean a hardware interrupt not handled by a higher-priority interrupt thread. In a properly constructed system, this usually can only be the (kernel-handled) timer interrupt (which has the highest interrupt priority) as other higher-priority interrupts normally have higher-priority handler threads.

the code that is executed when an interrupt occurs (Figure 5.1, bottom row). In other words, it supports my theory that it is probably RTLinux itself that impairs the system's interruptibility.

ACCURACY OF PREDICTIONS, VERIFICATION OF ASSUMPTIONS. The worst-case real-time–handler invocation times I observed (center column in Figures 5.1 and 5.2) seem to be in line with my expectations for L4RTL (39 µs vs. 42 µs), but not for RTLinux: The worst-case, total time I measured for RTLinux (56 µs) exceeds both my estimation ($29884 - 800 = 29084$ cycles $= 36.4$ µs) and, more significantly, the worst-case time I measured for L4RTL (39 µs). It seems that RTLinux has a larger cache footprint than L4RTL.

I verified my expectation that dirty cache lines do not impair predictability by setting up my cache flooder to flush, but not to dirty the caches. As I anticipated, all worst-case times (and even the shape of my density diagrams) stayed the same.

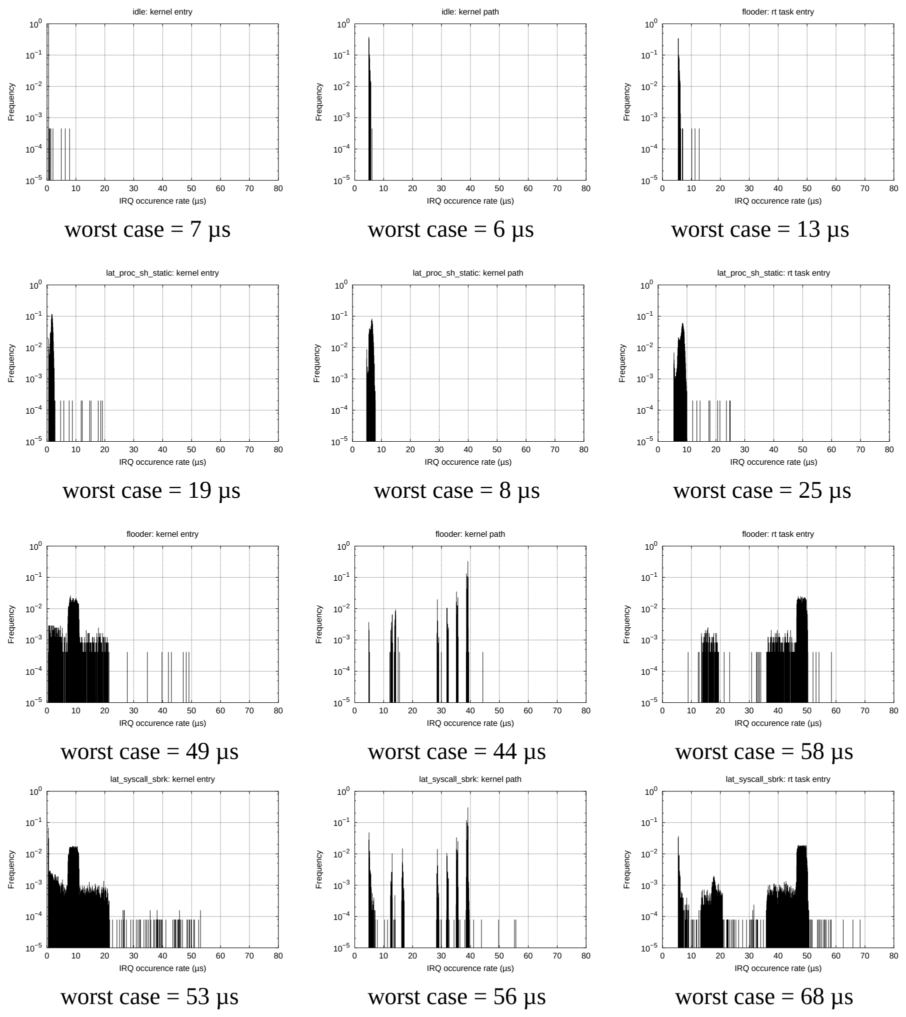
I verified my assumption that in Fiasco, the timer-interrupt handler is the main reason for interrupt delaying. I altered the experiment to measure under worst-case conditions the time the timer-interrupt handler disables interrupts. The result was 20 µs. This result suggests that Fiasco's timer-interrupt handling should be modified to leave interrupts enabled—by sacrificing either portability (by using a timer that has lower acknowledgment cost but is not available in older PCs, such as the Local APIC's timer interrupt) or accuracy (by allowing interruptions in the acknowledgment sequence).

AUXILIARY RESULTS. My measurements indicate that the main architectural difference between RTLinux and L4RTL—real-time tasks run in their own address spaces under L4RTL—introduces less uncertainty than blocked interrupts and caches. I deduce that providing address spaces for real-time tasks does not lead to unacceptable worst-case scheduling delays.

Another interesting result, even though not directly related to the subject of this thesis, is that the worst-case latency for RTLinux is 68 µs. It seems like RTLinux' authors did not take into account the penalty for accessing dirty cache lines when they claimed RTLinux guarantees 15-µs response times on current PC hardware [RTL01].

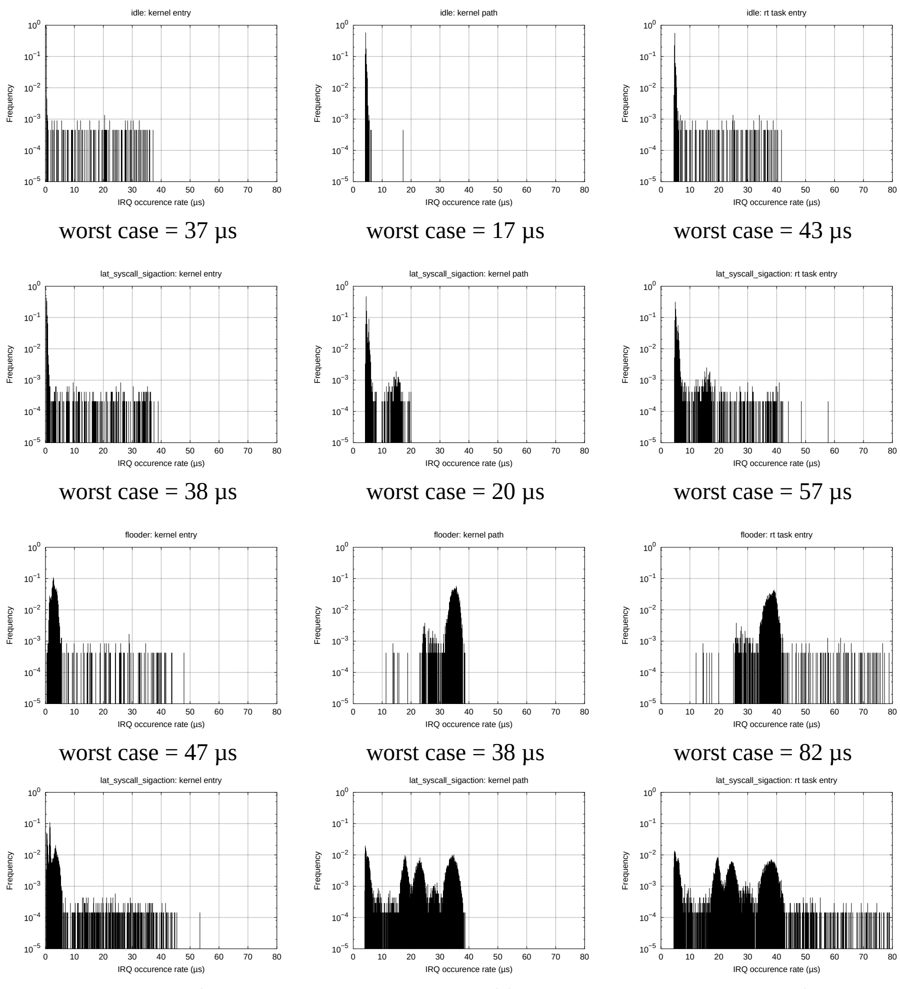
SUMMARY. I have evaluated the interruptibility of Fiasco, an operating system that I designed using Chapter 3's methodology and built using only nonblocking synchronization.

My results indicate that my methodology lead to an operating-system kernel with excellent interruptibility, comparable to a minimal real-time executive. Interrupt-response latency was low despite the presence of long critical sections (such as the task-delete operation). This was possible because my methodology allows only low-overhead lock-free synchronization (or very short critical sections) for global data needed during context switches, and maintains preemptability for all longer critical sections.



Legend: X axis shows interrupt latency. Y axis shows density of occurrence of particular latencies. Note that the Y axis has a logarithmic scale.

Figure 5.1: RTLinux performance under no load (top row), hbench (2nd row), cache flooder (3rd row), and hbench + cache flooder combined. Left: Time to enter kernel mode. Center: Time to activate handler in RTLinux thread. Right: Accumulated, total time



worst case = 37 µs | worst case = 17 µs | worst case = 43 µs

worst case = 38 µs | worst case = 20 µs | worst case = 57 µs

worst case = 47 µs | worst case = 38 µs | worst case = 82 µs

worst case = 53 µs | worst case = 39 µs | worst case = 85 µs

Legend: X and Y axes have the same meaning as in Figure 5.1

Figure 5.2: L4RTL performance under no load (top row), hbench (2nd row), cache flooder (3rd row), and hbench + cache flooder combined. Left: Time to enter kernel mode. Center: Time to activate handler in L4RTL thread. Right: Accumulated, total time

Chapter 6

Conclusion

6.1 Contributions of this work

This dissertation comprises three main contributions and a number of auxiliary contributions.

MAIN CONTRIBUTIONS.

Design methodology for real-time systems. (Chapter 3: Sections 3.1–3.3.3; Chapter 5)

I developed a pragmatic approach for developing real-time systems using non-blocking synchronization, with the following properties:

- My methodology leads to real-time systems with excellent real-time properties and a low synchronization overhead.
- It is applicable to both kernels and user-mode programs.
- It extends naturally to multiprocessor systems.
- It is not restricted to CPUs that provide an atomic CAS2 instruction.
- Using my methodology is straightforward. It makes use of interfaces similar to those of mutual exclusion and uses only simple lock-free data structures.

Low-overhead priority inheritance. (Chapter 3: Sections 3.3.1–3.3.4; Chapter 5)

I proposed wait-free lock designs for uniprocessor and multiprocessor kernels. I also discussed kernel interfaces that allow user-mode programs to take advantage of priority inheritance. My lock designs have the following properties:

- My lock designs provide a variant of priority inheritance, which I call “locking with helping.”
- My multiprocessor lock implements a new real-time resource-access protocol, the multiprocessor priority-inheritance protocol (MPIP).
- They provide synchronization semantics as powerful as those of wait-and-notify monitors, as long as *nested* monitor calls never block.

Real-time serializer. (Chapter 3: Section 3.3.5; Chapter 5: Section 5.1)

I proposed a simple modification to Massalin-Pu synchronization servers (also known as serializers) to make them applicable to real-time systems. This modification has the following properties:

- Its synchronization semantics are equivalent to locking with priority inheritance.
- It is possible to implement this scheme with acceptable performance, but in direct comparison to my implementation of priority inheritance, it has significantly higher overhead.

Fiasco microkernel. (Chapters 4, 5)

I verified my methodology in the design and implementation of the Fiasco microkernel. It has the following properties:

- Fiasco runs on uniprocessors and multiprocessors with x86 CPUs.
- It implements my "locking with helping" schemes in their uniprocessor and multiprocessor kernel variants.
- As Fiasco has been designed according to my design methodology, it is (almost) completely preemptible, which translates to excellent real-time properties.
- Fiasco does not need to sleep within any critical sections (nested or nonnested). This provides evidence that my locking scheme's restriction of disallowing nested critical section that may block is not a show stopper.

AUXILIARY CONTRIBUTIONS.

Address spaces in real-time systems. (Chapter 5: Section 5.2)

I provide evidence that the introduction of address spaces to real-time systems does not cause uncertainties larger than those generally accepted by system designers, namely jitter induced by caches and interrupt blocking.

Caches and worst-case estimations. (Chapter 5: Section 5.2)

The fact that the worst-case latency I measured for RTLinux is four times larger than what its designers claim shows that the effect of caches worst-case execution times are often underestimated.

6.2 Suggestions for future work

I see three major areas for future work.

First, in this thesis I have not discussed schedulability analysis for MPIP, as it does not fit into the scope of this work. However, work in this field has already started within our group.

Second, the balance between the use of lock-free and wait-free synchronization within the Fiasco microkernel certainly can be improved further. In my first design, I overestimated the cost of a number of locked operations and concluded wrongly that they could not be put into a *very short* critical section protected by disabling interrupts without causing noticeable priority inversion. I therefore created numerous critical section protected by wait-free locks in which both the locking overhead and cost of a kernel entry are higher than the cost of the critical section. As I write this conclusion, members of our group refine Fiasco using this insight. This work will result in a much better performing kernel.

Third, being able to give real-time guarantees on a multiprocessor machine depends on the condition that all CPUs get the machine resources (e. g., bus bandwidth) they need. This condition can be guaranteed only if the effects on these resources caused by (non-real-time) programs on all CPUs can be contained. The quantification of such effects and devising ways to their containment remains future work.

6.3 Concluding remarks

In this thesis I have described a pragmatic approach for constructing real-time systems that use only nonblocking synchronization. Taking recent theoretical contributions with a grain of salt and using a practical interpretation of what constitutes nonblocking synchronization, I developed a methodology for designing real-time system that is easy to use and yields excellent results in low synchronization overhead and real-time performance, especially interrupt latency.

Using my methodology, I developed the Fiasco microkernel, which is more than just an academic experiment. Because of its real-time properties, TU Dresden's operating-systems group uses Fiasco as the base of the Dresden Real-Time Operating System DROPS.

Bibliography

[ARJ97a] James H. Anderson, Srikanth Ramamurthy, and Rohit Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–238, Santa Barbara, California, 21–24 August 1997.

[ARJ97b] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions of Computer Systems*, 15(2):134–165, May 1997.

[Aud91] Neil C. Audsley. Resource control for hard real-time systems: A review. Technical Report YCS 159, University of York, Department of Computer Science, Real-time Systems Research Group, August 1991.

[Ber93] B. N. Bershad. Practical considerations for non-blocking concurrent objects. In Robert Werner, editor, *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, Pittsburgh, PA, May 1993. IEEE Computer Society Press.

[BF95] Peter A. Buhr and Michael Fortier. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[BS97] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, Seattle, WA, June 1997.

[Che95] Chia-Mei Chen. *Scheduling Issues in Real-Time Systems*. PhD thesis, University of Maryland, Institute for Advanced Computer Studies, 1995.

[Elp01] Kevin Elphinstone. Proposed L4 scheduling behavior to support RT system construction. In *Proceedings of the Second Workshop on Microkernel-based Systems*, Lake Louise, Banff, Canada, 2001.

[GC96] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 123–136, Berkeley, CA, USA, October 1996. USENIX.

[GDFR90] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *USENIX Summer Conference*, pages 87–96, Anaheim, CA, June 1990.

[Gre99] Michael Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.

[HBB$^+$98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[HHL$^+$97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of µ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[HHW98] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *5th Annual Australasian Conference on Parallel And Real-Time Systems (PART)*, Adelaide, Australia, September 1998.

[Hil92] D. Hildebrand. An architectural overview of QNX. In *1st USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[HJT$^+$93] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 94–105, Asheville, NC, December 1993.

[HLR$^+$01] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22th IEEE Real-Time Systems Sysmposium (RTSS)*, London, UK, December 2001.

[Hoh98] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`.

[Hoh02] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[How76] John H. Howard. Signaling in monitors. In *Second International Conference on Software Engineering*, pages 47–52, San Francisco, CA, October 1976.

[Int99] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, 1999.

[LH01] Jork Löser and Hermann Härtig. Cache influence on worst case execution time of network stacks. Technical Report TUD-FI02-07-Juli-2002, TU Dresden, July 2001.

[Lie95] J. Liedtke. On µ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[Lie96a] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[Lie96b] J. Liedtke. Toward real µ-kernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):40–61, 1973.

[LR80] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[Meh99] Frank Mehnert. L4RTL: Porting RTLinux API to L4/Fiasco. In *Workshop on a Common Microkernel System Platform*, Kiawah Island, SC, December 1999. Available from URL: `http://os.inf.tu-dresden.de/~fm3/l4rtl_l4ws.pdf`.

[MHSH01] Frank Mehnert, Michael Hohmuth, Sebastian Schönberg, and Hermann Härtig. RTLinux with address spaces. In *Proceedings of the Third Real-Time Linux Workshop*, Milano, Italy, November 2001.

[Moi97] M. Moir. Transparent support for wait-free transactions. *Lecture Notes in Computer Science*, 1320:305, 1997.

[MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275, New York, USA, May 1996. ACM.

[PAB$^+$95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 314–324, Copper Mountain Resort, CO, December 1995.

[PRS$^+$01] B. Pfitzmann, J. Riordan, Ch. Stüble, M. Waidner, and A. Weber. Die PERSEUS System-Architektur. In D. Fox, M. Köhntopp, and A. Pfitzmann, editors, *Verlässliche IT-Systeme (VIS)*. GI, Vieweg, September 2001.

[RM95] I. Rhee and G. R. Martin. A scalable real-time synchronization protocol for distributed systems. In *16th IEEE Real-Time Systems Symposium (RTSS)*, 1995.

[RSL88] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[RTL01] RTLinux FAQ. URL: `http://www.rtlinux.org/documents/faq.html`, October 2001.

[SB96] Vik Sohal and Mitch Bunnell. A real OS for real time — LynxOS provides a good, portable environment for embedded applications. *Byte Magazine*, 21(9):51, September 1996.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[Sun97] Jun Sun. *Fixed-Priority End-To-End Scheduling In Distributed Real-Time Systems*. PhD thesis, University of Illinois, Department of Computer Science, 1997.

[TNR90] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time Mach: Towards a predictable real-time system. In USENIX, editor, *Mach Workshop Conference Proceedings, October 4–5, 1990. Burlington, VT*, pages 73–82, Berkeley, CA, USA, October 1990. USENIX.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[Val95a] John D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, May 1995.

[Val95b] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, August 1995. Erratum available at `ftp://ftp.cs.rpi.edu/pub/valoisj/podc95-errata.ps.gz`.

[YB97] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, January 1997. The USENIX Association.

[ZPS99] K. M. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: a small-memory real-time microkernel. In *17th ACM Symposium on Operating System Principles (SOSP)*, pages 277–291, Kiawah Island, SC, December 1999.