Pragmatic nonblocking synchronization for real-time systems

Michael Hohmuth Hermann Härtig

Dresden University of Technology
Department of Computer Science
drops@os.inf.tu-dresden.de, http://os.inf.tu-dresden.de/drops/

Abstract

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

This paper reports novel results in various respects: We restrict the usage of lock-free mechanisms to cases where the widely available atomic single-word compare-and-swap operation suffices. We show how Brinch Hansen's monitors (alias Java's synchronized methods) can be implemented on top of our mechanisms, thereby demonstrating their versatility. We describe in detail how we used the mechanisms for a full reimplementation of a popular microkernel interface (L4). Our kernel—in contrast to the original implementation—bounds execution time of all operations. We report on a previous implementation of our mechanisms in which we used Massalin's and Pu's single-server approach, and on the resulting performance, which lead us to abandon this well-known scheme.

Our microkernel implementation is in daily use with a user-level Linux server running a large variety of applications. Hence, our system can be considered as more than just an academic prototype. Still, and despite its implementation in C++, it compares favorably with the original, highly optimized, non-real-time, assembly-language implementation.

1 Introduction

In recent years, nonblocking data structures have 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 interesting data structures in a nonblocking fashion. Others have conceived general methodologies for transforming any algorithm into a nonblocking one; however, these results have a more theoretical nature as the methodologies often lead to very inefficient implementations. The next section briefly discusses a number of these works.

In contrast to this boom, we know of only a few system implementations that successfully exploit nonblocking synchronization. The only two operating systems we are aware of that use exclusively nonblocking synchronization are SYNTHESIS [16] and the CACHE kernel [7].

One of the problems with the approach is that it appears difficult to apply to many modern CPU architectures: 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 paper, we present a pragmatic approach for building nonblocking real-time systems. Our methodology works well even on CAS2-less archi-

tectures. It does not rely solely on lock-free synchronization for implementing nonblocking data structures—which would be both inconvenient and slow on the architectures we considered. Instead, our methodology does allow for locks, but ensures that the system is wait-free nonetheless. In addition, our technique is easy to apply because from a developer's perspective, it looks much like programming with mutual exclusion using monitors.

We describe the application of our approach to build a real system: Using our methodology, we developed the Fiasco microkernel, a kernel for the DROPS real-time operating system [8] that runs on x86 CPUs. This kernel is an implementation of the L4 microkernel interface [15], and it is sufficiently mature to support all the software developed for L4, including DROPS servers and L^{4}Linux [9].[1] We evaluate the effectiveness of our methodology for nonblocking design by examining the Fiasco microkernel's real-time properties and synchronization overheads.

Fiasco currently runs only on uniprocessors. Consequently, we concentrate on single-processor implementation details. However, our methodology lends itself to multiprocessor-system implementations as well, and we point out routes for multiprocessor extensions.

We also discuss a number of nonblocking synchronization mechanisms. In their SYNTHESIS work, Massalin and Pu [16] introduced the concept of a "single-server" thread (a variant of the "serializer" pattern first described by Hauser and associates [10]), which serializes complex object updates that cannot be implemented in a nonblocking fashion. In this paper, we present a simple modification to the single-server scheme that makes it truly nonblocking and useful for use in real-time systems. Furthermore, we 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 better performance.

[1] L^{4}Linux is a port of the Linux kernel (version 2.2.x) that runs as a user program on top of L4 and is binary compatible with original Linux.

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

This paper is organized as follows: In Section 2, we consider related work on nonblocking synchronization. In Section 3, we develop our methodology for designing wait-free real-time systems. Section 4 shows how we applied this methodology to the development of the Fiasco microkernel. In Section 5, we present performance values for the Fiasco microkernel, and we evaluate the kernel's real-time properties. In Section 6, we derive conditions for the applicability of our methodology for the development of multithreaded user-mode real-time programs. We conclude the paper in Section 7 with a summary and suggestions for future work.

2 Nonblocking synchronization and related work

2.1 Lock-free and wait-free synchronization

Overview. Nonblocking synchronization strategies have two important properties: First, they provide full preemptability and allow for multi-CPU concurrency. 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.

The concepts discussed in this section are not new in any way, and many systems implement variants of them such as optimistic concurrency control [1] and priority inheritance [20]. We describe them here for completeness.

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. During helping, *A* lends *B* its priority to ensure that no other, lower-prioritized activities can interfere. When *B* has finished, *A* executes its own critical section.

Wait-free object implementations satisfy a stronger form of block-freedom than lock-free synchronization (discussed in the next paragraph) as they guarantee freedom from starvation. Therefore, many authors 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.

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. If needed, retries can be delayed with an exponential backoff to avoid retry contention.[2]

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

Preconditions for using lock-free synchronization are that primitives for atomic memory modifications are available, and data is stored in type-stable memory management. We do not digress into type-stable memory management in this paper (see [7] for a discussion of operating-systems–related issues); the rest of this subsection discusses atomic memory modification.

[2] Backoff is never needed on single-CPU systems.

Atomic memory update. The x86 CPUs have two kinds of atomic memory-modification operations: a test-and-set instruction (TAS) and a CAS instruction. Newer models (Intel Pentium and newer) also have a double-size–word (8 bytes) compare-and-swap instruction (CASW). However, these CPUs do not support atomically updating two independent memory words (two-word compare-and-swap, CAS2).

A number of data structures can be implemented 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. [21, 18]

Valois introduced a lock-free single-linked list design supporting insertions and deletions anywhere in a list, as well as several other data structures [23, 22]. These designs also work with just CAS. However, Greenwald [6] 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 single-linked and double-linked lists. [6]

A number of techniques exist for implementing lock-free and wait-free general multi-word compare-and-swap (MWCAS) on top of CAS and CAS2, enabling nonblocking synchronization for arbitrarily complex data structures [11, 19, 2, 6]. 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.

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. The disadvantage of this method is (of course) that it does not work on multiprocessors.

Bershad [4] 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 [7] discuss a generalization of this technique to implement CAS2 or MW-CAS. 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.

Another technique to facilitate complex object updates is the "serializer" or "single-server" approach [10]. 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 Nonblocking synchronization in operating systems

We know of two other operating system projects that have explored nonblocking synchronization in the kernel: the CACHE kernel [7] and SYNTHESIS [16].

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 operating-system kernels.

Massalin and Pu [16] 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 [7] in their CACHE kernel. We will revisit the single-server approach in Section 4.4.

Greenwald and Cheriton [7] 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 paper, we will investigate how nonblocking systems can be implemented in such an environment.

2.3 Nonblocking synchronization vs. real-time systems

Nonblocking object implementations are of interest for real-time systems because they provide preemptability and avoid priority inversion. However, while it is well-known that wait-free method implementations are bounded in time (there is only a fixed number of threads we have to help; no retry loop), it is not immediately apparent that this also applies to lock-free synchronization. On the surface, lock-free methods (like the ones in Figure 3 in Section 4.3) look dangerous because of their potentially unlimited number of retries.

Fortunately, Anderson and colleagues [3] 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.

3 A design methodology for real-time systems

3.1 Design goals

Our main design goal was to allow for good real-time properties of our systems. More specifically, we wanted higher-priority threads to be able to preempt the system (including the kernel) at virtually any time, as soon as they are ready to run—thus allowing for good schedulability of event handlers

[12]. This should be true for sets of threads that depend on common resources, but even more so for independent thread sets.

Secondary goals to the first one are that short critical sections working on global state should induce essentially no overhead for synchronization; also, the synchronization scheme should work for both single-processor and multi-processor architectures.[3]

Finally, the design should be applicable to x86-compatible uniprocessors, that is, it must be implementable without CAS2.

3.2 Design guidelines

The first design goal rules out any synchronization scheme that suffers from priority inversion. Therefore, we have been looking 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. We therefore generally disallow lock-based schemes for frequently-used global state except where we have no other way out.

In particular, our design methodology comprises the following guidelines:

We classify a system's objects as follows: *Local state* consists of objects used only by related threads, that is, threads that cooperate on a given job or assignment. *Global state* consists of the objects shared by unrelated threads.

Frequently-accessed global state should be implemented using data structures that can easily be accessed with lock-free synchronization.

In Section 2.1, we mentioned a number of data structures that can be synchronized in this fashion on x86 CPUs using only CAS: Counters, bitfields, stacks, and FIFO queues.

[3] We will point out incompatibilities of our design methodology with multiprocessor architectures where they occur.

With the x86 CPU lacking anything better than single-word CAS, we suggest that other global data (like double-linked lists) are also implemented in a lock-free fashion, based on a software implementation of MWCAS.

In a kernel, the atomic update can be protected by disabling interrupts as discussed in Section 2.1. Of course, disabling interrupts does not help on multiprocessors; there, we suggest using spin locks to protect very short critical sections.

We discuss software-MWCAS for user-mode programs in Section 6.

Global state not relevant for real-time computing, and local data can be accessed using wait-free synchronization. We propose a wait-free priority-inheritance locking mechanism that can be characterized as "locking with helping," explained in more detail in Section 3.3. This kind of synchronization has some overhead. Therefore, it should be avoided for objects that otherwise independent threads must access.

In our synchronization scheme, waiting for events inside critical sections is not allowed. This restriction ensures wait-freedom. We will show in Section 3.3 that this restriction does not limit the synchronization mechanism's power.

Once a designer has decided which object should be synchronized with which scheme, our 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.

3.3 Wait-free locking with helping

We suggest 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 its critical section. 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 (because the previous time slice has been consumed, or because of some other reason), 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 high-priority thread 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.[4] This property ensures that the highest-priority threads get their critical sections through first. It makes our locking mechanism an implementation of priority inheritance.

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. We describe our solution in Section 4.3.

As long as critical sections do not nest, it is easy to see that our construction can be used to implement wait-and-notify monitors[5] [14] (or their recent descendant, Java synchronized methods). Whenever a monitor-protected object's method is called, we acquire the object's lock. The wait operation would then be implemented as an unlock–sleep–lock sequence. Figure 2 shows a possible monitor implementation that uses a simple lock-free semaphore, shown in Figure 1.

Synchronization is more difficult when more than one object can be locked at a time. We 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 our 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, we have chosen the first option; in fact, we constructed the kernel so that critical sections never need to sleep. We discuss synchronization in the Fiasco microkernel in more detail in Section 4.3.

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.

[4] 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 are beyond the scope of this work.

[5] There is a large variety of monitors with differing semantics, but most of them can be shown to have equivalent expressive power [13, 5]. Wait-and-notify monitors, also classified as "no-priority nonblocking monitors" [5], have first been used in Mesa [14].

```
class Binary_semaphore
{
  Thread_list 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 1 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 3.

```
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 2 Pseudocode for a wait-and-notify monitor based on a helping lock. This is a simple textbook implementation—except that it uses only non-blocking 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.

4 Synchronization in the Fiasco microkernel

We 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 [8]. The microkernel runs on uniprocessor x86 PCs, and it is an implementation of the L4/x86 binary interface [15]. It is able to run L^{4}Linux [9], 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/drops/`.

The kernel closely follows the design outlined in Section 3. In this section, we report how various data structures are synchronized in this kernel, and we detail the design of our wait-free locking-with-helping mechanism.

4.1 Kernel objects

Let us begin by briefly describing the objects the Fiasco microkernel implements. (For a philosophical discussion on what a microkernel should and should not implement, we refer to Liedtke [15].)

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 4.3).

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 a user-level handler thread. The kernel sends this thread a message every time the interrupt occurs.

Mapping trees. Like L4, the Fiasco microkernel allows transferring persistent 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, mappings depending on it are recursively flushed as well [15]. 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 double-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 start and end of the list.

Array of address space references. This array is indexed by an 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.

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 grows or shrinks. 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 Synchronization of kernel objects

Following our design methodology from Section 3.2, the global state should be synchronized using lock-free synchronization while for local state the overhead of wait-free locking is acceptable. Primarily, we closely adhered to these guidelines. But we also made the requirements somewhat stronger where performance is critical, and we allowed a small relaxation where it did not affect real-time properties.

Local state. Threads are the most interesting objects that must be synchronized. We accomplish synchronization using wait-free locks (described in Section 4.3). However, for IPC-performance reasons we do not lock all of a thread's state. Instead, we 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 double-linked list of other threads that want to send the thread a message. The state word can be synchronized using CAS. For the double-linked sender list we use a simulated MWCAS that disables interrupts during memory modification.[6]

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, we 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.

We 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 double-linked present and ready lists, we had to resort to simulate MWCAS by disabling interrupts for a short time. These lists and the sender list mentioned previously were the only objects for which we had to revert to this "ugly" but inevitable synchronization method.[7]

We believe that is it unnecessary to implement the kernel allocators for pages and mapping trees with lock-free synchronization; here we used wait-free locking, as for the local state. We allowed this relaxation of our guidelines in these instances for the following reason: 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 our assertion become untrue in the future, we will revisit this design decision.

4.3 Wait-free locking in the Fiasco microkernel

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

[6] For the prospective port of the kernel to SMP machines, we plan to protect this MWCAS using a spin lock per receiver.

[7] 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 seldom and can be synchronized using a spin lock.

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

First, thread locks in the Fiasco microkernel 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 wins, but the Fiasco microkernel's IPC system call semantics dictate that the receiver gets the CPU first. The hint is a flag that 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 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.

Second, when locking other objects (including threads), 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 push to-be-deleted threads out of their critical sections.

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, we did not find this limitation to be very restricting. We completely avoided nesting critical sections that might want to sleep: We 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.

Figure 3 shows pseudocode for our sleep and wakeup operations. As a means to avoid race conditions between sleep and wakeup, we 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 our 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.

4.4 Single-server synchronization revisited

Before we implemented the wait-free locking scheme described in Section 4.3, we experimented with Massalin's and Pu's single-server synchronization scheme discussed in Section 2.1. In this section, we discuss how the single-server mechanism can be changed for real-time systems, and why we 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 our helping-lock design from Section 3.3, we 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 we first designed and implemented our wait-free synchronization mechanism, we drew inspira-

```
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 3 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.

tion from Massalin's and Pu's work. In particular, our design looked as follows:

Our 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.

We consider this design to be not inelegant, but unfortunately, it required a context switch for every locked operation. Only later we realized that this mechanism in fact shares many properties with the wait-free locking scheme with priority inheritance we derived in Section 3.3. Our new locking mechanism is less complex and performs much better than our original single-server scheme.

5 Performance evaluation

To evaluate the real-time properties of the Fiasco microkernel and the overhead of its synchronization mechanisms, we conducted two series of measurements. First, to verify that the kernel matches our requirements with regards to preemptability and scheduling, we measured the lateness of a user-level interrupt handler. Second, we measured the overhead of our synchronization primitives in a number of microbenchmarks.

System	Max. lateness
Fiasco µ-kernel / L^{4}Linux	65 µs
L4/x86 / L^{4}Linux	541 µs
RTLinux	58 µs

Table 1 Maximum lateness of a periodic 250-µs interrupt handler. On the Fiasco µ-kernel and on L4/x86, the handler ran in a user task of its own; in RTLinux, the handler was a real-time task running in kernel mode.

We carried out these measurements on a 200 MHz Pentium Pro machine. The CPU's built-in local APIC served as the interrupt source.

5.1 Real-time characteristics

For this test, we set up a timer device to trigger a hardware interrupt every 250 µs. We created a user-level task containing a high-priority handler thread connected to the interrupt, and we measured the time between interrupt occurrences. From the results, we computed the maximum lateness. During measurements, a cache-flooding application and a Linux system running various multi-user benchmarks ran concurrently with the handler thread, inducing a high load on the system.[8]

We conducted this test on three operating systems: on the Fiasco microkernel with L^{4}Linux, on Liedtke's high-performance L4/x86 kernel with L^{4}Linux, and, for comparison, on RTLinux [24] (with the handler running in kernel mode). Table 1 shows the maximum latenesses for the three systems. (The average lateness was very small—smaller than 1 µs on all systems.)

It turns out that maximum lateness in the Fiasco microkernel is an order of magnitude smaller than that for L4/x86. That is because L4/x86 uses interrupt disabling liberally throughout the kernel to synchronize accesses to kernel data structures. Moreover, the Fiasco microkernel is close to RTLinux even though the interrupt handler under RTLinux runs in kernel mode and in the kernel's address space while Fiasco handlers run in their own task.

[8] These measurements are equivalent to those Mehnert [17] carried out in 1999. Mehnert's results showed a much worse maximum lateness for the Fiasco microkernel; these poor results were caused by a kernel bug that has since been fixed.

System	Cycles [P5]	Cycles [P-II]
counter, unsynchronized	2	2
counter, CAS	13	12
counter, Fiasco thread lock	245	245
counter, old Massalin–Pu–style thread lock (includes one context switch)	627	607
IPC	653	810
IPC, L4/x86	398	438
IPC, L4/x86, small addr. space[a]	184	300

[a] L4/x86 offers an optimization called "small address spaces," which significantly reduces context-switch cost for small address spaces by implementing it using a segment switch instead of a page-table switch [9].

Table 2 Synchronization overhead (under no contention) in the Fiasco microkernel on two different machines. For comparison, we show IPC times (one-way) for the Fiasco microkernel and for L4/x86.

We measured the numbers in the P5 column on a 133-MHz Pentium box, and the number in the P-II column on a 400-MHz Pentium-II box. We used normal C or C++ programs (not hand-optimized assembly) to conduct the measurements.

5.2 Microbenchmarks

We carried out a small series of measurements to evaluate the overhead of our synchronization mechanisms, and to get clues for future optimizations.

We implemented a simple one-word counter and protected its increment operation using the following synchronization schemes: CAS; a wait-free helping lock (Fiasco microkernel's new synchronization); and wait-free object lock with the operation running in a different thread (Fiasco microkernel's old Massalin–Pu single-server–style synchronization). For comparison, we measured an unprotected counter, and a complete address-space-

crossing short-IPC operation in the Fiasco microkernel (needs no lock), and we put all results into relation with the performance of Liedtke's L4/x86's IPC performance. Table 2 shows the results.

We are quite satisfied with the performance overhead of our new helping-lock implementation. Even though we are yet to optimize our code, we have already experienced a more-than-twofold improvement in comparison to the implementation of Massalin's and Pu's single-server scheme.

6 Nonblocking synchronization in user-mode programs

In this section we discuss how our design methodology can be applied to multithreaded user-level programs.

Let us recall three preconditions for the effectiveness of our methodology for nonblocking design: First, MWCAS can only be simulated if concurrent access to the shared data can be disabled. Second, to ensure wait-freedom, critical sections protected by priority-inversion–safe locks must not block. Third, helping only works if the threading system provides priority inheritance. Meeting these conditions for user-level programs is most definitely possible, but can be difficult. We discuss the conditions in turn.

The interrupt-disabling method to prevent preemptions does not work on user level. Therefore, disabling concurrent access implies some kind of locking. As critical sections accessing data that is updated using simulated MWCAS are typically very short, priority inversion is best prevented by employing preemption-safe locks (i. e., locks that prevent descheduling a lock-holding thread in favor of a thread that shares the lock-protected data structure). In general, the locking mechanism depends on the underlying operating system. For example, spin locks can be used on multiprocessor systems that always gang-schedule all of the program's threads; uniprocessors can use the operating-system–assisted MWCAS implementation we discussed in Section 2.1, or an operating-system–assisted preemption-safe lock.

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 paging. For that, user programs need operating-system support.

Optimally, the operating system should support priority inheritance in the kernel.

In summary, multi-threaded user programs can use our design technique if the operating system provides some support that real-time systems provide frequently, or can easily implement: MWCAS support, preemption-safe locking, memory pinning, and priority inheritance.

7 Summary and conclusion

We introduced a pragmatic methodology for designing nonblocking real-time systems that is not dependent on an atomic memory-modification primitive like CAS2; just CAS is sufficient.

Our methodology consists of four basic guidelines: (1) partition the system into global and local objects; (2) implement the global objects using lock-free synchronization as far as possible; (3) protect the other objects using locks with priority inheritance; (4) never wait for events inside critical sections. We argued that following these rules ensures wait-freedom.

We derived three conditions for an operating system on which our methodology becomes applicable for wait-free user-mode programs: (1) the operating system must provide help for a user-mode implementation of MWCAS, either directly or by supporting preemption-safe locks; (2) it must provide a service for memory pinning; (3) it must support priority inheritance.

We proposed a wait-free locking-with-helping mechanism with priority inheritance, and we showed that it is similar in effect to, but better

performing than, the single-server synchronization mechanism introduced by Massalin and Pu [16]. We devised a monitor implementation that works on top of our locking mechanism.

The application of our methodology can lead to systems with excellent real-time properties: We have built the Fiasco microkernel using the methodology. Together with L^{4}Linux, the Fiasco microkernel reaches a level of preemptability that is close to that of RTLinux.

Currently our work has two significant limitations. First, our performance results are preliminary in many ways. Our next steps will be to analyze in more detail what is causing worst-case interrupt latencies of more than 50 µs, and to look at processor dependencies. From this evaluation, we intend to develop a model for predicting worst-case interrupt latencies for our methodology. Second, we have not compared the performance of our synchronization primitives to the performance of primitives found in other commercial and research operating systems. Both limitations are being addressed as part of the first author's thesis work.

In the near future, we plan to add multiprocessor support to the Fiasco microkernel in order to verify our methodology for multiprocessors. Following which, we plan to optimize the Fiasco microkernel's locking-with-helping mechanism and thread switching.

Also, we plan to research the applicability of the techniques for user-level programs in more depth, that is, with real software and measurements.

Availability

The Fiasco microkernel is freely available; researchers are invited to study the implementation of our design methodology, and to experiment with it.

Fiasco and L^{4}Linux can be downloaded from `http://os.inf.tu-dresden.de/drops/`.

Acknowledgements

We would like to thank Frank Mehnert for providing his measurement framework, and Michael Peter for improving Fiasco's synchronization primitives. We are grateful to our shepherd, Sheila Harnett, and to our anonymous reviewers for their valuable suggestions.

Special thanks go to Thomas Roche who helped debugging our prose.

This project has been partially funded by the Deutsche Forschungsgemeinschaft in the framework of the Sonderforschungsbereich 358, and supported by generous grants from IBM (University Partnership and Shared University Research programs) and from Intel (MRL Lab Hillsboro).

References

[1] Atul Adya, Barbara Liskov, Robert Gruber, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of SIGMOD*, San Jose, CA, May 1995.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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`.

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

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

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

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

[17] Frank Mehnert. L4RTL: Porting RTLinux API to L4/Fiasco. In *Workshop on a Common Microkernel System Platform*, Kiawah Island, SC, December 1999.

[18] 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.

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

[20] 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.

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

[22] 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, 2–23 August 1995. Erratum available at `ftp://ftp.cs.rpi.edu/pub/valoisj/podc95-errata.ps.gz`.

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

[24] 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.