Helping in a multiprocessor environment

Michael Hohmuth Michael Peter

Dresden University of Technology
Department of Computer Science
D-01062 Dresden, Germany
email: fiasco-core@os.inf.tu-dresden.de

Abstract

This report presents Fiasco-SMP, a port of the Fiasco microkernel to the multiprocessor-x86 architecture. We the discuss design principles we used, and the resulting design for remote-thread manipulation in Fiasco. In particular, we show how we extended Fiasco's implementation of priority inheritance to fit a multiprocessor environment.

Our design 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.

Moreover, we propose an extension to the L4 interface that allows server threads to specify that the kernel is allowed to schedule them on a remote CPU (i. e., not on their home CPU) when they become runnable after an IPC. We believe that this behavior has advantages for an important class of servers as it cuts out IPI latency from the server-startup delay.

1 Wait-free locking with helping

Fiasco is an implementation of the L4 microkernel interface [2]. It is a real-time microkernel for x86 CPUs mainly developed by this report's first author.

Fiasco aims for low-latency thread activation when interrupts or timeouts occur, and bounded worst-case execution times. It accomplishes these goals by being completely preemptible, and by synchronizing its kernel data structures using solely nonblocking synchronization. Fiasco uses lock-free synchronization for simple operations on global data structures (such as the ready queue), and wait-free synchronization for more complex operations such as thread manipulation and IPC handshake.

As a wait-free synchronization primitive, Fiasco implements a CPU-time donation scheme known as "locking with helping." Helping occurs when a thread A wants to lock an object that is already locked by another thread B. Instead of blocking, A donates time to B, helping it to finish its critical section. Helping is an implementation of priority inheritance.

In [1], we described an efficient implementation of locking with helping for uniprocessor systems. However, we only hinted on how to extend helping for a multiprocessor environment. Indeed, there are a number of interesting problems:

- Where should thread-manipulating locked operations execute—on the locker's CPU or on the locked thread's CPU?
- Where should helping occur on multi-CPU systems—on the CPU of the helper, or on the CPU of the current lock owner?
- If a thread is runnable after it has been locked, should it be scheduled on the CPU of the previous lock owner, on its previous CPU, or on any other CPU?

In this report, we propose answers to these questions based on design principles we applied when we designed Fiasco's SMP support.

This report is organized as follows. In Section 2, we give an overview over basic assumptions we made when we designed Fiasco's SMP support, and derive design principles for a multiprocessor kernel. Section 3 describes Fiasco's SMP thread-locking mechanism in detail, and we discuss thread lockdown, wakeup, and helping. We conclude the report in Section 4 with an overview of accomplished and remaining work.

2 Fiasco's SMP execution model

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

- Prefer CPU-local data structures when 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.

The most important data structures 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.

Fiasco currently implements CPU-local ready queues. (Other data structures potentially accessed during a context switch are hardware-interrupt descriptors and timeout queues. The former are local to the thread attached to the interrupt, and therefore local to the interrupt handler's CPU.[1] We haven't converted the latter to a CPU-local data structure yet. Fortunately, they are outside the scope of this report.)

Static CPU binding. Systems that dynamically schedule user-mode threads on multiple CPUs run the danger of cache pollution: Second-level cache working sets continuously have to be exchanged between CPUs, slowing down applications and increasing the likelihood of cache-capacity 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.

Fiasco follows this belief. It binds threads to a "home CPU." Migration only occurs on user request with an interface similar to the one Völp proposed in [3].

However, Fiasco supports temporary remote execution of a thread's in-kernel part. This feature facilitates helping, which we explain in Section 3.3. 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, Fiasco uses this feature 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. 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 (aka local locking): Thread A runs the operation on B's CPU. Thread B is locked on its own CPU.

Local execution (aka 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, we 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 us to exclude caching effects from further consideration.

Fiasco implements the second variant, local execution. Locked operations usually execute on the locker's CPU (i. e., except if helping occurs—see Section 3.3). In the uncontented case, Fiasco's remote-locking implementation uses a single compare-and-swap (CAS) operation. If the CAS fails (because the thread is currently running or because another thread owns the lock), Fiasco falls back to remote notification (to lock down a running thread) or helping (in case the thread is already locked). We explain remote locking in detail in Section 3.

In Fiasco, locked operations never need to *synchronously* notify the locked thread's CPU. These operations 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 the locked thread becomes runnable. We describe the wakeup mechanism in detail in Section 3.2.

[1] Fiasco routes hardware interrupts to the home CPU of the interrupt handler.

3 Locking remote threads

In this section, we look at Fiasco's remote-locking mechanism in detail. We explore design alternatives and explain the choices we have made for Fiasco.

Locking remote threads raises the questions of dealing with threads that currently execute on another CPU (lockdown) and with wakeups. We discuss these issues in Sections 3.1 and 3.2. In Section 3.3, we discuss cross-CPU helping—the mechanism that provides system-wide priority inheritance.

3.1 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").

Fiasco implements 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.

In Fiasco, 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[2]; see Section 3.3). This IPI causes an immediate reschedule on B's CPU. Meanwhile, A polls B's status, waiting for B to be deactivated.

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. Fiasco secures 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 an randomized exponential backoff, before it retries the operation.

3.2 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 on 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.

Fiasco's multiprocessor solution is based on the CPU-local data structures and static CPU binding principles: It never runs unlocked kernel code (or user code) on a CPU other than a thread's home CPU. Instead, it 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]

Alternative: Wakeup binding. Let us discuss an alternative approach: Not enforcing the static CPU binding principle upon wakeup. Instead, if the thread has a higher priority than the locker, immediately run the thread on the locker's CPU ("Wakeup binding").

This method has two interesting benefits for servers that usually answer within the same time slice, such as a small name server or even L^{4}Linux: First, there is no latency induced by IPIs, and second, it implies that the kernel can use its "fast local IPC path" to deliver short messages.

There are a number of drawbacks with this approach. A small but obvious drawback is that user code can not anymore assume strict priority order of execution, and synchronization schemes that rely on priority order (such as the one currently used in L^{4}Linux) will fail.

The major drawback, of course, is the cache pollution problem the static CPU binding principle was intended to solve. However, for certain types of servers such as very small servers or frequently-used system-level servers like L^{4}Linux, this may not be a problem at all.

Clearly, there is a tradeoff between IPC latency and cache-pollution cost. Therefore, we propose to make wakeup binding an optional, user-controllable feature of L4.

3.3 Helping a remote thread

Helping is an implementation of priority inheritance. It avoids priority inversion by donating CPU time of high-priority threads that want to acquire a lock to low-priority lock holders, effectively pushing the low-priority thread out of its critical section and preventing a mid-priority thread from blocking the high-priority thread.

Priority inheritance is desirable even across CPU boundaries: We want to avoid situations in which a mid-priority thread on one CPU prevents a high-priority thread on another CPU from running. Therefore, we explored ways to

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

[3] Ready-queue removal does not need to be signaled as Fiasco uses a lazy ready-queue–update discipline.

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 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 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. We prefer local helping's behavior.

Apart from semantics, remote helping is less preferable also because it is more complex to implement (and therefore, has a higher run-time cost): Like remote locking (see Section 2), it must deal with the thread-migration problem: At the time thread A arrives at B's CPU, thread B might have migrated elsewhere.

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, Fiasco implements local helping.

Let us now 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 that wants to help another thread B on a different CPU finds that B is already running on that CPU? We considered two alternatives:

Sleep and callback. Thread A registers a callback IPI with thread B's 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.

Both 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, we went with the polling method.

There is a fixed order in which helping (or polling) threads acquire a lock: Helpers enqueue in the lock's wait queue, or rather "helper queue," which is sorted by global priority,[4] 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 A has been helped and has executed its critical section on a CPU different from its home CPU (the "guest CPU"), which thread should run on that CPU once A leaves its critical section?

If thread A 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 A leaves its critical section. This is the highest-priority thread that was waiting for the lock. It can be equivalent with thread A'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 A 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 guest CPU is also not an option for thread A.

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

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

Once thread A has been descheduled from its guest CPU, it becomes runnable on its home CPU again. In Fiasco, no special notification is necessary: A 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 considers it automatically. Also, if A now is the highest-priority thread of its home CPU, that CPU's

[4] Fiasco synchronizes accesses to a lock's helper queue using a spin lock.

scheduler will poll, waiting for A being removed from guest CPUs, and run it immediately.

4 Summary and conclusion

In this report, we discussed design choices we made for Fiasco's SMP implementation, and we developed Fiasco's remote-thread–locking mechanism, in particular the remote-helping mechanism.

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

From these principles, we 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.

We discussed wakeup binding, an alternative to static CPU binding for one special case: When a thread is runnable after is was locked, there are cases in which it is advantageous to immediately start it on the CPU on which it was locked, instead of on its home CPU. In particular, certain types of servers, such as very small servers or L^4Linux, can benefit if IPC behaved this way. We proposed to make this behavior an optional, configurable feature of the L4 interface.

In the near future, we plan to quantize the effect of wakeup binding to allow developers to assess this binding scheme's applicability to their projects.

References

[1] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[2] 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.

[3] M. Völp and J. Liedtke. Threads on an L4/x86 SMP nucleus. In *Proceedings of the First Workshop on Common Microkernel System Platforms*, Kiawah Island, SC, USA, December 1999.