Non-Blocking Synchronization in Operating Systems

Fabian Bläse
Friedrich-Alexander University Erlangen-Nurnberg (FAU)
fabian.blaese@fau.de

ABSTRACT

Synchronizing shared objects using blocking techniques can come with significant performance and reliability issues, like deadlocks. Non-blocking algorithms can offer a solution for these problems by allowing multiple threads to modify shared data structures concurrently and atomically committing the result. However, despite this obvious advantage, their use in modern operating systems is quite rare. Still, quite a bit of research has gone into using non-blocking data structures in operating systems. This paper presents concepts for using non-blocking algorithms in the context of operating systems and discusses their advantages and disadvantages. Both a straightforward methodology for wait-free locking with helping and a cooperative non-blocking scheduler are covered.

1 INTRODUCTION

A big contributor to the complexity of operating system development is the correct and performant synchronization of shared data structures. Typically, blocking synchronization is used to prevent multiple threads from simultaneously accessing these data structures. While synchronizing using locks might seem straightforward at first, a lot of well-known problems arise from blocking synchronization. The most common problems are deadlocks, livelocks, and starvation, which can be hard to identify, especially with a high level of parallelism. Apart from that, blocking synchronization allows only a single thread to make progress. All other threads have to wait until the lock is released. Depending on the granularity of the lock, this can hurt the performance of algorithms significantly.

Non-blocking algorithms try to solve many of these problems by implementing algorithms in a way that does not require threads to block when accessing shared data structures. In the late 1990s and early 2000s, a lot of research went into non-blocking data structures. Many frequently used data structures have been designed and implemented in a non-blocking fashion.

However, even today, their use in operating systems is quite rare, despite seemingly having many clear advantages over the currently used synchronization mechanisms. One of the problems of non-blocking synchronization is the relatively high complexity of the actual algorithm compared to blocking synchronization. A lot of effort is required to correctly design non-blocking data structures while also not sacrificing too much performance. Also, many of the already existing implementations require instructions for updating multiple memory locations atomically, which are not available on prevalent architectures like x86.

In this paper, I want to explore a few of the non-blocking concepts that have been researched in the context of operating systems. First, I introduce the basic concepts and different types of non-blocking synchronization, including a short summary of the common ABA problem. Also, I shortly introduce fundamentals necessary for implementing and evaluating specific non-blocking algorithms. After that, I describe three different concepts of non-blocking operating systems, which have already been researched. The actual implementation of these concepts of the respective authors is discussed afterwards. Finally, I evaluate different aspects of the discussed concepts. This includes evaluation of performance, portability, and applicability, and focuses on the advantages and disadvantages compared to blocking algorithms.

2 BASIC CONCEPTS

It is necessary to be familiar with a few basic concepts for implementing non-blocking algorithms before the actual ideas of non-blocking operating systems can be discussed, so I shortly introduce them. First, I summarize two different types of non-blocking synchronization: lock-free synchronization and wait-free synchronization. After that, I introduce cooperative scheduling. Also, the problem of priority inversion is briefly addressed.

2.1 Non-Blocking Synchronization

Non-blocking synchronization comprises techniques to synchronize parallel procedures without having to block other threads accessing the same data structures. There are different kinds of algorithms, which can be separated into the following three categories: non-blocking synchronization, lock-free synchronization, and wait-free synchronization. Lock-free and wait-free synchronization are both subsets of non-blocking synchronization. The following differentiation is inspired by [5].

Lock-free synchronization describes algorithms where the use of locks is completely avoided. Instead, modifications to shared memory locations are made in parallel, but the result is only committed if the value has not been changed since it has been read and modified. If the value has not changed, the result can be written to the memory location. If a different routine has modified the value concurrently, the operation is repeated using the new value. Because this test-and-write operation has to be done atomically, lock-free algorithms require hardware that supports compare-and-swap (CAS) instructions. While support for the simple CAS instruction is widespread, more complex algorithms sometimes require atomic modifications to multiple not necessarily contiguous memory locations. This requires double compare-and-swap (DCAS) or even multi-word compare-and-swap (MWCAS) instructions, whose support is not nearly as widespread as CAS. While software simulation of MWCAS instructions is possible using DCAS or even only unary CAS [2, 3, 11], the use of DCAS and especially MWCAS should be avoided, if performance on hardware that does not support DCAS natively is of interest.

On the other hand, **wait-free synchronization** characterizes algorithms that ensure that waiting is never required when entering critical code sections. Instead, a thread helps the holder of the lock to finish its work in the critical section. By lending the thread's priority to the currently working thread, the critical section can be executed with the priority of the thread trying to acquire the

lock, so the release of the lock is not dependent on a lower priority thread. On top of the characteristics of lock-free synchronization, Wait-free synchronization guarantees starvation-freedom.

2.2 ABA Problem

Non-blocking synchronization suffers from the so-called ABA-problem. When a thread tries to update a memory location using non-blocking algorithms, it usually first reads the current value from memory into a local variable, and writes the updated value back into the shared location if the shared location still has the same value. Otherwise, the operation is repeated. However, it might happen that the value of the shared variable is concurrently updated multiple times during the update of the local copy, so in the end it has the same values as before. When the thread now compares the shared variable with its local copy, the concurrent update goes unnoticed because their values are identical.

This is particularly likely to happen with algorithms where the shared variable is a pointer to a memory location, which might be reused [10]. If multi-word CAS instructions are available, this problem can be solved by additionally incrementing a modification counter [10, 14]. However, hardware implementations of DCAS are not very widespread and software simulated MWCAS has significant performance disadvantages, as previously discussed.

2.3 Cooperative Scheduling

Task schedulers are a fundamental part of all operating systems. With the appearance of multi-core processors, their algorithms have got even more complex because concurrent access to shared data must also be protected against concurrent access from multiple processing cores. Most modern operating systems implement task scheduling using a preemptive mechanism, where timer interrupts are used to regularly trigger switches between tasks. As a result, tasks can be interrupted by the scheduler at any time. Contrary to that, cooperative scheduling does not initiate task switches based on elapsed time, but only when the scheduler is called explicitly. While this has the advantage that threads cannot be preempted at any time, it requires programs to reliably call the scheduler at regular intervals, especially when executing loops or when encountering anything that could halt execution indefinitely, like locks, because otherwise monopolization of the processing unit is possible.

While this might sound like a significant disadvantage at first glance, it also comes with clear advantages. First of all, it is possible to use a scheduler like this on systems without support for time-based interrupts, which are necessary for a preemptive scheduler. Another advantage is that task switches only happen at distinct positions, which are known at compile-time. This not only has the advantage of very simple task switches, because it is not necessary to save and restore the state of a processor core and the implementation of task switches requiring less platform-specific code, but also allows for certain assumptions for other parts of the code, where no task switches can occur [14]. These assumptions will be discussed and used to our advantage later on.

2.4 Priority Inversion

In an operating system using priority-based scheduling, priority inversion can occur when a higher priority thread has to wait

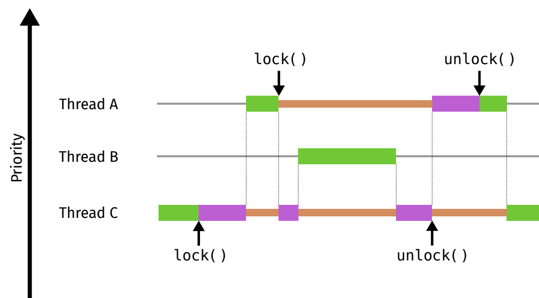

Figure 1: Example for priority inversion. Progress of the high priority thread A is hindered by the medium priority thread B, which has a higher priority than thread C (currently in possession of the lock).

for resources, which a lower priority thread currently owns. This can happen, for example, if a semaphore is used to protect data structures which are in use by both high and low priority threads. While a lower priority thread is in possession of the semaphore, a higher priority thread trying to acquire the same semaphore is effectively prevented from being scheduled because it has to wait for the semaphore to become free. The significance of this issue gets a lot clearer if a medium priority thread is added. Because medium priority threads can prevent low priority threads from getting execution time, a high priority thread waiting for the semaphore currently blocked by the low priority thread is unable to proceed, even though it has a higher priority than all currently executed threads. In Figure 1, this issue is shown with three threads of different priorities. In the beginning, the low priority thread C is executed. After it enters a critical section by acquiring a lock, it is interrupted by the high priority thread A, which then also tries to acquire the lock. Because thread A no has to wait for the lock to become free, thread C is scheduled again. However, thread B launches and interrupts the execution due to its higher priority, even though the high priority thread A still tries to acquire the lock. The execution of thread A might be delayed indefinitely, as long as the medium priority thread B is executed.

3 NON-BLOCKING OPERATING SYSTEM CONCEPTS

Research has been made to solve the issues associated with blocking data structures in operating systems. Next, I want to introduce three of the resulting non-blocking concepts. First, I will address wait-free locking with helping proposed by Hohmuth and Härtig, which is a non-blocking mechanism to prevent priority inversion. Afterwards, implicit cooperative non-blocking scheduling and the resulting simplifications of lock-free implementations discovered by Negele et al. will be discussed.

3.1 Wait-Free Locking with Helping

Hohmuth and Härtig propose a wait-free scheduler methodology, which avoids the issue of priority inversion while still allowing the use of locks [5]. The main idea is that instead of blocking until a resource is available, the thread trying to acquire it passes the CPU to the thread which currently occupies the resource. Hence,

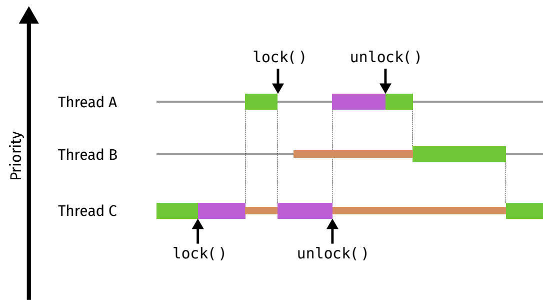

Figure 2: Priority inheritance using wait-free locking with helping. Thread A lends its priority to thread C until the lock is freed. Therefore, the medium priority thread B can not interrupt the low-priority thread C until the critical section is left.

the thread in possession of the resource can continue its work and free the lock as fast as possible, even though it might have a low priority. As soon as the lock is not required anymore, the thread frees the lock and passes the CPU back to the helper thread. An additional data structure to remember which threads are currently trying to acquire the lock is necessary, so the CPU can be passed back to the thread with the highest priority. To ensure that this helping mechanism is actually wait-free, a thread is not allowed to sleep or wait as long as it holds a lock. Otherwise, the release of the lock might be delayed indefinitely.

Implementing this helping scheme has one major advantage to blocking and passing the CPU to the scheduler: Because the critical section of the thread in possession of the lock is effectively executed with the priority of the thread trying to acquire the lock, priority inversion is avoided. To ensure that the CPU is passed back to the thread with the highest priority, a stack is used. If a thread is only preempted by other threads with a higher priority, the items on the stack are ordered by their priority, so the thread with the highest priority always is on top of the stack. Therefore, the highest priority thread gets the lock first, which makes this methodology an implementation of priority-inheritance [15]. It is noteworthy that this assumption is only true for single-threaded CPUs, as Hohmuth points out, because on multithreaded CPUs, lower priority threads might get scheduled on a different processor core. In that case, either a different data structure like a priority queue is necessary, or the thread trying to acquire the lock has to be migrated to the same processing core as the lock owner, to ensure correct ordering on the stack.

Figure 2 shows how this mechanism solves the problem of priority inversion compared to Figure 1. When the high priority thread A tries to acquire the lock, it passes the CPU to thread C, which currently holds the lock, so the critical section of thread C is executed with the higher priority of thread A. Therefore, the medium priority thread B cannot preempt the execution of the critical section. After releasing the lock, the CPU is passed back to thread A because it is on top of the helper stack.

While this idea still uses locks, still suffers from all problems associated with them and, depending on the definition of non-blocking algorithms, is not strictly wait-free, it ensures progress of the critical section when acquiring the locks, at least if critical sections do not sleep or wait, so priority inversion is avoided and system-wide throughput is guaranteed.

3.2 Implicit Cooperative Non-Blocking Scheduling

A concept introduced by Negele, Friedrich, Oh, and Egger is implicit cooperative non-blocking scheduling [14]. As previously discussed in Section 2.3, cooperative scheduling, in contrast to preemptive scheduling, has the advantage of not requiring any special hardware support. Instead, cooperative scheduling mechanisms can be implemented entirely software-driven in a high-level programming language. However, for it to work as expected, all programs must call the scheduler in regular intervals to prevent monopolization of the CPU. This means that correct scheduling behavior is dependent on the correct implementation of user-space programs.

To solve this issue, Negele et al. propose a special compiler feature, which inserts the calls to the scheduler at appropriate positions automatically. By this, the cooperativeness of programs can be ensured [1]. To be able to control the frequency of scheduler calls, especially around tight loops, a mechanism to determine how much time has passed is necessary. Negele et al. decided to use a software instrumented instruction counter for this, mainly to stay portable and keep the added overhead as small as possible. In addition to scheduler calls, instructions are inserted to decrement a counter called quantum by the number of instructions emitted and therefore executed since the last scheduler call. The scheduler is only called if this counter reaches zero. By this, the amount of instructions executed before the scheduler is called can be defined. When the scheduler switches to a thread, the quantum is initialized with the maximum amount of instructions a thread is allowed to execute before it has to call the scheduler. By reserving a hardware register for this quantum, the footprint of those instructions can be kept minimal. It has been shown that the overhead of software instrumented instruction counters is acceptable [9]. Of course, by counting instructions, the actual time passed until the scheduler is called is dependent on the target. Especially with modern architectures, the time required to execute complex instructions is nearly impossible to predict reliably.

By using cooperative scheduling, all points at which context switches can happen are known at compile time. This not only simplifies context switches, but also allows for certain optimizations to lock-free algorithms, one of which will be discussed in the next section.

3.3 Unbounded Lock-Free Scheduler Queue

Unbounded queues are required for a scheduler to store the processes, which are ready to be scheduled. As the scheduler is a very critical part of the operating system and is executed quite often, it makes sense to implement these queues in a lock-free fashion. An approach for this is made by Negele et al. [14].

Because unbounded queues can contain arbitrary amounts of items, they require some form of memory allocation. This is typically done by allocating memory for every element which shall be stored in the queue. When implemented naively, new memory is allocated on every insertion and deallocated on every removal,

which has an undesirable performance disadvantage. To speed up insertion and removal of elements, Negele et al. tried to make the reuse of already allocated list elements possible. However, the reuse of memory allocations significantly increases the likelihood of the ABA problem discussed in Section 2.2 because identical pointer values are used for different queue elements. Negele et al. point out that this might also happen after multiple memory allocations and deallocations because previously deallocated memory can be reused by the allocator. According to their work, even pointer tagging, which uses parts of the pointer value to store a tag, so pointers to identical memory locations can be differentiated, does not completely avoid the ABA problem.

A solution for this problem called "Hazard pointers" has been introduced by Maged Michael [10]. These pointers store references that are currently in use by a thread doing a queue operation. Therefore, memory locations which are pointed to by hazard pointers are not safe to reuse. This can happen, for example, when one thread tries to modify the tail element to enqueue a new element. If the tail element gets dequeued, reused, and enqueued while the current thread tries to insert its own element, the CAS operation might succeed even though the tail element has since been modified because the same memory location is stored in the tail pointer. This problem is solved by storing the current value of the pointer, which shall later be used in the CAS operation. Other threads trying to reuse the allocation first have to check if no hazard pointers reference it.

However, hazard pointers are required for every task that might still use the allocation. Without any additional restrictions, this means that hazard pointers are required for every thread of an operating system. Because every thread could be executing a task switch, hazard pointers must be stored in a thread-local storage. The number of hazard pointers to which reused memory locations have to be compared to, is therefore not limited. Because a pointer first has to be compared against all hazard pointers, the amount of comparisons and therefore the execution time of the scheduler would be dependent on the number of currently running threads. Negele et al. discovered that it is possible to take advantage of characteristics of cooperative scheduling to solve this problem. By guaranteeing that the queueing operation of a task switch is uncooperative, the maximum amount of task switches that can be executed in parallel is limited by the amount of processor cores. As a result, hazard pointers can be stored in a processor-local data structure with a fixed size, and the amount of comparisons remains unchanged during the execution.

4 IMPLEMENTATIONS

The non-blocking concepts just introduced have been utilized in operating system implementations by their respective authors. These implementations and their particularities will be examined in the following section.

4.1 Fiasco Kernel

The Fiasco Kernel is a microkernel introduced by Michael Hohmuth and Hermann Härtig for the DROPS operating system, which implements non-blocking synchronization concepts [5]. The microkernel shall be a drop-in replacement for the previously used L4 microkernel and therefore must implement the L4 microkernel interface [8]. One of the reasons for implementing a new microkernel has been the suboptimal real-time properties of the L4 microkernel, which only uses interrupt disabling for synchronization [7]. One of the main design goals has been that higher priority threads cannot be blocked by lower priority threads, so priority inversion is impossible [4]. Fiasco uses a preemptive scheduler, so user-space applications are not required to take care of the scheduler calls. At the time of release, Fisco had no support for multithreading, but it was added later on [6].

Hohmuth and Härtig considered local and global state objects separately. On the one hand, local state objects are only used by threads belonging together. These objects are generally synchronized using the wait-free locking mechanism introduced in Section 3.1, but with some exceptions due to IPC-performance reasons. These exceptions are instead synchronized with lock-free algorithms using CAS instructions. Because DCAS is not available on the intended target platform, software simulated MWCAS is used where necessary.

On the other hand, global state objects, which are shared between unrelated threads, are primarily synchronized using lock-free synchronization. The performance of this synchronization mechanism is crucial, as it influences the properties of real-time threads. Like with local state objects, lock-free algorithms were implemented using simple CAS instructions, with one notable exception: The double-linked present and ready lists require multi-word CAS for lock-free synchronization, which the target platform does not support, so simulated MWCAS with interrupt disabling was used instead. Hohmuth and Härtig point out that this does not pose a problem when multi-processor support is added because the ready list is processor-local, and the present list is only accessed rarely. As a simplification, kernel allocators for pages and mapping trees have been implemented using the wait-free locking mechanism. Hohmuth and Härtig argue that this is not a problem for real-time threads, as they do not allocate memory or resize mapping trees, so access to these shared resources is not nesessary once the thread has been initialized.

4.2 Scheduler of the Native Kernel

A different approach for the implementation of non-blocking synchronization in operating systems is presented by Negele et al.. The scheduler of their "Native" kernel is implemented using only lock-free synchronization. Their main goal was to reduce complexity and platform-specific code to improve portability, which has been achieved by using implicit cooperative multitasking and completely avoiding the use of locks. Using lock-free synchronization also has the benefit of avoiding problems associated with blocking synchronization like deadlocks, starvation, and others.

While most operating systems nowadays use preemptive multitasking, Native uses cooperative multitasking. Modifications to the compiler of the operating system have been made to allow automatic insertion of scheduler calls, as discussed in Section 3.2, so the scheduler calls are entirely transparent to the application developer. Using a cooperative scheduler not only has the advantage of requiring less platform-specific code, as timer interrupts do not have to handle context switches at arbitrary locations, but also allows clever optimization of lock-free algorithms by preventing

task switches during their execution, which has been discussed in Section 3.3. Native makes use of this optimization to achieve a lock-free implementation of the scheduling queue with constant time and space overhead using only simple CAS instructions.

5 EVALUATION

Now that the concepts and implementations have been discussed, their authors evaluated the approaches for their respective performance, portability and simplicity, and applicability.

5.1 Performance

First, I will take a look at the performance of the presented approaches. For the Fiasco kernel, especially real-time characteristics have been a focus. To evaluate the latency of interrupt handling, Hohmuth and Härtig measured the lateness of a user-level interrupt handler [5]. This was done by triggering a hardware interrupt every 250 µs, which is handled by a high-priority handler thread, while concurrently running a cache-flooding application and multi-user benchmarks. The time difference between handlers is measured. The results are compared to Liedtke's L4 kernel [8] and RTLinux [16]. The test was run on a 200 MHz Pentium Pro machine with interrupts being generated by the build-in local APIC. The results of this test can be seen in Table 1. While these are no real-time constraints, the maximum lateness using the Fiasco kernel is a lot smaller than using the L4 kernel, which uses interrupt disabling to synchronize access to shared data structures. The Fiasco kernel is very close to the maximum lateness of RTLinux, which is quite impressive, considering that the handler is run in kernel mode on RTLinux, whereas Fiasco handlers require a task switch.

The Native kernel, on the other hand, is primarily designed to improve simplicity and portability and to avoid problems associated with blocking synchronization. Performance has not been the primary focus but still is important. Therefore, Negele et al. ran a few benchmarks and compared their performance to Windows Server 2008 R2 and Linux 2.6.32 on a 64-core x86_64 machine [14]. First, a microbenchmark was run to evaluate the time required to create, schedule, and destroy a thread. Figure 3 shows the average time required for the creation of a thread with an empty body. Due to the lightweight implementation of threads, which is possible due to cooperative scheduling, Native has a clear advantage in this respect. Figure 4 shows the average time required for a task switch. The threads only call the scheduler in a loop in this benchmark. Context switches in Native are quite a lot slower than in Linux. Negele et al. state that this is due to the single global ready queue, which causes significant contention.

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 the handler for a periodic 250µs interrupt on a 200 MHz Pentium Pro machine. The handler is executed in user mode on Fiasco and L4/x86. Taken from [5].

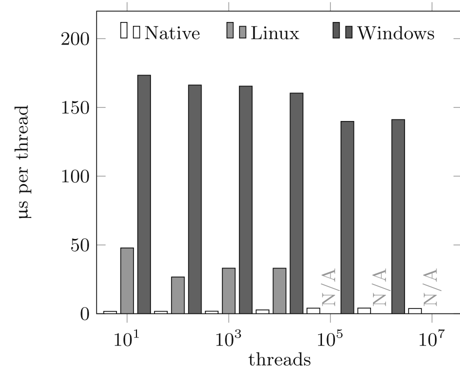

Figure 3: Average creation time of an empty thread. Taken from [14].

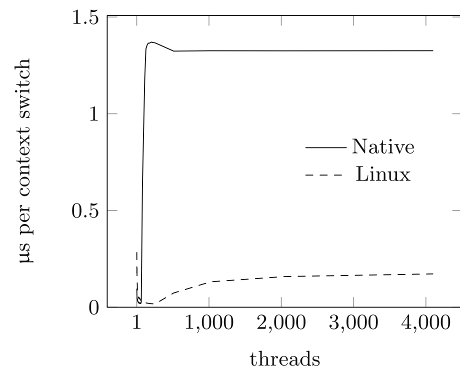

Figure 4: Average context switch time of empty tasks. Taken from [14].

Apart from these microbenchmarks, Negele et al. measured the performance of a few real-world applications, which is especially interesting, as these also show the impact of the software instruction counter. For this, a matrix multiplication benchmark is used. The absolute runtime of this benchmark using one thread was 14.7s for Native and 12.5s for Linux. According to Negele et al., this difference is mostly caused by the quantum checks. However, the compiler does not contain any optimizations. However, they expect this performance difference to get insignificant when using compiler optimizations.

5.2 Portability and Simplicity

A very interesting aspect of non-blocking operating systems is portability and simplicity. As already described in Section 2.1, non-blocking algorithms are, apart from generally requiring CAS instructions, hardware independent. Hence, much more code, like the scheduler, can completely be written in a high-level programming language. This means that operating system code can be much simpler and shorter, and porting the kernel to a new architecture requires less code modification.

Component	Lines	Ratio
Interrupt Handling	299	75%
Memory Management	357	20%
Modules	82	17%
Multiprocessing	215	37%
Runtime Support	174	12%
Scheduler	547	37%
Total	1674	27%

Table 2: Code lines of Native in comparison to A2. Taken from [13].

In his doctoral thesis [13], Negele compared the number of code lines of the Native kernel to the A2 kernel, which has been an earlier multiprocessor kernel approach of the Native Systems Group at ETH Zurich [12]. The results of this comparison are shown in Table 2. The lock-free scheduler of Native discussed in Section 4.2 comprises only 547 lines of code, which is a significant improvement being only 37% as much as the scheduler implementation of A2. Also, Negele states that only six functions had to be rewritten to port Native to a new platform. This is an important advantage over traditional operating systems with blocking synchronization, where, more often than not, significant changes to essential parts of the operating systems have to be made.

5.3 Applicability

With those clear advantages regarding portability and simplicity, one might be curious why non-blocking operating systems are not ubiquitous. To answer this question, the applicability of operating systems using non-blocking synchronization has to be evaluated.

First, I will consider Native's cooperative scheduling approach, which is required to implement the scheduler queue efficiently. While Negele et al. have shown that the necessary explicit scheduler calls can be hidden from the programmer by using a special compiler, which makes cooperative scheduling behave mostly like preemptive scheduling, it is still necessary to compile all programs using this special compiler, so the source code has to be available. Also, system stability is fully dependent on the correctness of the compiler of all user-space programs. This makes a cooperative approach unsuitable for operating systems, where the developer is not under full control of all software. Furthermore, without the guarantees made by cooperative scheduling, the implementation of an unbounded lock-free queue is a lot more difficult, especially without the availability of DCAS instructions. However, this approach still is very suitable for certain embedded systems, especially considering its portability and simplicity, which have been discussed before.

On the other hand, Hohmuth and Härtig's approach using preemptive scheduling does not suffer from the same issues arising from the cooperative scheduler used in Native. Nevertheless, the implementation still requires locks for certain data structures, which can be implemented wait-free, but still suffer from some of the issues associated with locking synchronization. Other than that, the approach is promising because it has relevant advantages over operating systems using blocking synchronization.

6 CONCLUSION

Non-blocking synchronization can have significant advantages over traditional synchronization techniques. It has been shown that their use in an operating system is not only possible but also comes with some notable advantages. Especially real-time applications and operating systems with hard priorities can benefit from the improvements made. Still, the actual implementation is quite challenging, even though for many common data structures, non-blocking algorithms already exist, because these have requirements, which are hard to fulfill. Despite that, the use of non-blocking synchronization in operating systems should be considered due to its clear advantages.

REFERENCES

[1] Melvin E. Conway. 1963. Design of a Separable Transition-Diagram Compiler. *Commun. ACM* 6, 7 (July 1963), 396–408. https://doi.org/10.1145/366663.366704

[2] Michael Greenwald. 1999. *Non-Blocking Synchronization and System Design*. Ph.D. Dissertation. Stanford, CA, USA.

[3] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-Word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, Berlin, Heidelberg, 265–279.

[4] Michael Hohmuth. 1998. The Fiasco Kernel: Requirements Definition. Available at http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz.

[5] Michael Hohmuth and Hermann Härtig. 2001. Pragmatic Nonblocking Synchronization for Real-Time Systems. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 217–230.

[6] Michael Hohmuth and Michael Peter. 2001. *Helping in a Multiprocessor Environment*. Technical Report.

[7] Hermann Härtig, Michael Hohmuth, and Jean Wolter. 1998. Taming Linux. In *In Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*. 49–56.

[8] J. Liedtke. 1995. On micro-kernel construction. *ACM SIGOPS Operating Systems Review* 29 (12 1995), 237–250. https://doi.org/10.1145/224056.224075

[9] J. M. Mellor-Crummey and T. J. LeBlanc. 1989. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS III)*. Association for Computing Machinery, New York, NY, USA, 78–86. https://doi.org/10.1145/70082.68189

[10] Maged Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on* 15 (07 2004), 491 – 504. https://doi.org/10.1109/TPDS.2004.8

[11] Mark Moir. 1997. Transparent Support for Wait-Free Transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG '97)*. Springer-Verlag, Berlin, Heidelberg, 305–319.

[12] Pieter Johannes Muller. 2002. *The active object system design and multiprocessor implementation*. Ph.D. Dissertation. ETH Zurich, Zürich. https://doi.org/10.3929/ethz-a-004453415 Diss., Technische Wissenschaften ETH Zürich, Nr. 14755, 2002.

[13] Florian Negele. 2014. *Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System*. Ph.D. Dissertation. https://doi.org/10.3929/ETHZ-A-010335528

[14] Florian Negele, Felix Friedrich, Suwon Oh, and Bernhard Egger. 2017. On the Design and Implementation of an Efficient Lock-Free Scheduler. 22–45. https://doi.org/10.1007/978-3-319-61756-5_2

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. 1990. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.* 39, 9 (1990), 1175–1185. https://doi.org/10.1109/12.57058

[16] Victor Yodaiken and Michael Barabanov. 1997. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*. USENIX Assocication, Anaheim, CA.