LFTHREADS: A lock-free thread library

Anders Gidenstam*
School of Business and Informatics
University of Borås
SE-501 90 Borås, Sweden.
anders.gidenstam@hb.se

Marina Papatriantafilou
Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden.
ptrianta@chalmers.se

Abstract

This extended abstract presents LFTHREADS*, a thread library entirely based on lock-free methods, i.e. no spin-locks or similar synchronization mechanisms are employed in the implementation of the multithreading. Since lock-freedom is highly desirable in multiprocessors/multicores due to its advantages in parallelism, fault-tolerance, convoy-avoidance and more, there is an increased demand in lock-free methods in parallel applications, hence also in multiprocessor/multicore system services.* LFTHREADS *is the first thread library that provides a lock-free implementation of blocking synchronization primitives for application threads; although the latter may sound like a contradicting goal, such objects have several benefits: e.g. library operations that block and unblock threads on the same synchronization object can make progress in parallel while maintaining the desired thread-level semantics and without having to wait for any "slow" operations among them. Besides, as no spin-locks or similar synchronization mechanisms are employed, memory contention can be reduced and processors/cores are able to do useful work. As a consequence, applications, too, can enjoy enhanced parallelism and fault-tolerance. For the synchronization in* LFTHREADS *we have introduced a new method, which we call* responsibility hand-off *(RHO), that does not need any special kernel support. The RHO method is also of independent interest, as it can also serve as a tool for lock-free token passing, management of contention and interaction between scheduling and synchronization. This paper gives an outline and the context of* LFTHREADS*. For more details the reader is refered to [7] and [8].*

[a]This work was done while the author was a PostDoc at Algorithms and Complexity, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

1 Introduction

Multiprogramming and threading allow the processor(s) to be shared efficiently by several sequential threads of control. Here we study synchronization issues and algorithms for realizing standard thread-library operations and objects (create, exit, yield and mutexes) based entirely on *lock-free* methods.

The rationale in LFTHREADS is that active processors or cores should always be able to do useful work when there are runnable threads available , regardless of what other processors/cores do; i.e. despite others simultaneously accessing shared objects related with the implementation of the LFTHREADS-library and/or suffering stop failures or delays (e.g. from I/O or page-fault interrupts).

Even a lock-free thread library needs to provide blocking synchronization objects, e.g. for mutual exclusion in legacy applications and for other applications where threads might need to be blocked, e.g. to interact with some external device. Our new synchronization method in LFTHREADS implements a mutual exclusion object with the standard blocking semantics for application threads but *without enforcing mutual exclusion among the processors* executing the threads.We consider this an important part of the contribution. It enables library operations blocking and unblocking threads on the same synchronization object to make progress in parallel, while maintaining the desired thread-level semantics, without having to wait for any "slow" operation among them to complete. We achieved this via a new synchronization method, which we call *responsibility hand-off* (RHO), which may also be useful in lock-free synchronization constructions in general, e.g. for *token-passing, contention management and interplay between scheduling and synchronization*. The method is lock-free and manages thread execution contexts without needing special kernel or scheduler support.

Related and motivating work A special kernel-level mechanism, called *scheduler activations*, has been pro-

posed and studied [2, 5], to enable user-level threads to offer the functionality of kernel-level threads with respect to blocking and also leave no processor idle in the presence of ready threads, which is also the goal of LFTHREADS. It was observed that application-controlled blocking and interprocess communication can be resolved at user-level without modifications to the kernel while achieving the same goals as above, but multiprogramming demands and general blocking, such as for page-faults, seem to need scheduler activations. The RHO method and LFTHREADS complement these results, as they provide thread synchronization operation implementations that do not block each other unless the application blocks within the same level (i.e. user- or kernel-level). LFTHREADS can be combined with scheduler activations for a hybrid thread implementation with minimal blocking.

To make the implementation of blocking mutual exclusion more efficient, operating systems that implement threads at the kernel level may split the implementation of the mutual exclusion primitives between the kernel and user-level. This is done in e.g. Linux [6] and Sun Solaris [27]. This division allows the cases where threads do not need to be blocked or unblocked, to be handled at the user-level without invoking a system call and often in a non-blocking way by using hardware synchronization primitives. However, when the calling thread should block or when it needs to unblock some other thread, an expensive system call must be performed. Such system calls contain, in all cases we are aware of, critical sections protected by spin locks.

Although our present implementation of LFTHREADS is entirely at the user-level, its algorithms are also well suited for use in a kernel - user-level divided setting. With our method a significant benefit would be that there is no need for spin locks and/or disabling interrupts in either the user-level or the kernel-level part.

Further research motivated by the goal to keep processors busy doing useful work and to deal with preemptions in this context includes: mechanisms to provide some form of control on the kernel/scheduler to avoid unwanted preemption (cf. e.g. [18, 16]) or the use of some application-related information (e.g. from real-time systems) to recover from it [4]; [3] and subsequent results inspired by it focusing on scheduling with work-stealing, as a method to keep processors busy by providing fast and concurrent access to the set of ready threads; [25] aims in a similar direction, proposing thread scheduling that does not require locking (essentially using lock-free queuing) in a multithreading library called Lesser Bear; [32] studied methods of scheduling to reduce the amount of spinning in multithreaded mutual exclusion; [33] focuses on demands in real-time and embedded systems and studies methods for efficient, low-overhead semaphores; [1] gives an insightful overview of recent methods for mutual exclusion.

There has been other work at the operating system kernel level [21, 20, 11, 12], where basic kernel data structures have been replaced with lock-free ones with both performance and quality benefits. There are also extensive interest and results on lock-free methods for memory management (garbage collection and memory allocation, e.g. [31, 23, 22, 9, 10, 14]).

The goal of LFTHREADS is to implement a common thread library interface, including operations with blocking semantics, in a lock-free manner. It is possible to combine LFTHREADS with lock-free and other non-blocking implementations of shared objects, such as the NOBLE library [29] or software transactional memory constructions (cf. e.g. [19, 26]).

2 Preliminaries and problem description

System model The system consists of a set of processors or cores, each of which may have its own local memory as well as it is connected to a shared memory through an interconnect network. The shared memory supports atomic read and write operations of any single memory word, and also stronger single-word synchronization primitives, such as Compare-And-Swap (CAS) and Fetch-And-Add (FAA) These primitives are either available or can easily be derived from other available primitives [17, 24] on contemporary multicore and/or multiprocessor architectures.

Lock-free synchronization *Lock-freedom* [13] is a type of non-blocking synchronization that guarantees that in a set of concurrent operations at least one of them makes progress each time operations interfere and thus some eventually completes. The correctness condition for atomic non-blocking operations is *linearizability* [15], which guarantees that even when operations overlap in time, each of them appears to take effect at an atomic time instant within its time duration, such that the effect of each operation is consistent with the effect of a corresponding operation in a sequential execution in which the operations appear in the same order as the order of these time instants.

Non-blocking synchronization is attractive as it offers advantages over lock-based synchronization, w.r.t. priority inversion, deadlocks, lock convoys and fault tolerance. It has also been shown, using well-known parallel applications, that *lock-free* methods imply at least as good performance as lock-based ones in several applications, and often significantly better [28, 30].

The problem and LFTHREADS's API The LFTHREADS library defines the following procedures for thread han-

dling[a]:

create(thread,main): creates a new thread which starts in the procedure main; *exit*: terminates the calling thread and if this was the last thread of the application/process the latter is terminated too;

yield: causes the calling thread to be put on the ready queue and the (virtual) processor running it to pick a new thread to run from the ready queue.

For blocking mutual exclusion-based synchronization between threads, LFTHREADS provides a mutex object supporting the operations:

lock(mutex): attempts to lock the mutex. If it is locked already the calling thread is blocked and enqueued on the waiting queue of the mutex;

unlock(mutex): unlocks the mutex if there are no waiting threads in the waiting queue, otherwise the first of the waiting threads is made runnable and becomes the owner of the mutex (only the thread owning the mutex may call *unlock*);

trylock(mutex): tries to lock the mutex. Returns true on success, otherwise false.

3 Blocking thread synchronization and the RHO method

To facilitate blocking synchronization among application threads, LFTHREADS provides a mutex primitive, mutex_t. While the operations on a mutex, *lock*, *trylock* and *unlock* have their usual semantics for application threads, they are lock-free with respect to the processors in the system. This implies improved fault-tolerance properties against stop and timing faults in the system compared to traditional spin-lock-based implementations, since even if a processor is stopped or delayed in the middle of a mutex operation all other processors are still able to continue performing operations, *even on the same mutex*. However, note that an application thread trying to lock a mutex is blocked if the mutex is locked by another thread. A faulty application can also dead-lock its threads. It is the responsibility of the application developer to prevent such situations.[b]

The *lock* operation locks the mutex and makes the calling thread its owner. If the mutex is already locked the calling thread is blocked and the processor switches to another thread. The blocked thread's context will be activated and executed later when the mutex is released by its previous owner.

In the ordinary case a blocked thread is activated by the thread releasing the mutex by invoking *unlock*, but due to fine-grained synchronization, it may also happen in other ways. In particular, note that checking whether the mutex is locked and entering the mutex waiting queue are distinct atomic operations. Therefore, the interleaving of thread-steps can e.g. cause a thread A to find the mutex locked, but later by the time it has entered the mutex queue the mutex has been released, hence A should not remain blocked in the waiting queue. The "traditional" way to avoid this problem is to ensure that at most one processor modifies the mutex state at a time by enforcing mutual exclusion among the processors, e.g. by using a spin-lock. In the lock-free solution proposed here, the synchronization required for such cases is managed with the *responsibility hand-off* (RHO) method proposed here. In particular, the thread/processor releasing the mutex is able, using appropriate fine-grained synchronization steps, to detect whether such a situation may have occurred and, in response, "hand-off" the ownership (or responsibility) for the mutex to some other processor.

By performing a *responsibility hand-off*, the processor executing the *unlock* can finish this operation and continue executing threads without waiting for the concurrent *lock* operation to finish (and vice versa). As a result, the mutex primitive in LFTHREADS tolerates arbitrary delays and even stop failures inside mutex operations without affecting the other processors' ability to do useful work, including operations on the same mutex. Roughly speaking, the RHO method handles cases where processors need to perform sequences of atomic actions on a shared object in a consistent and lock-free manner, for example a combination of (i) checking the state of a mutex, (ii) blocking if needed by saving the current thread state and (iii) enqueuing the blocked thread on the waiting queue of the mutex; or a combination of (i) changing the state of the mutex to unlocked and/or (ii) activating a blocked process from the waiting queue if there is any. "Traditional" ways to do the same use locks and are therefore vulnerable to processors failing or being delayed, which the RHO method is not.

The details of the RHO method are given in [7] and [8] together with a description of the other components of the library and an analysis including correctness, fault-tolerance and an experimental study on contemporary multiprocessor/multicore platforms.

4 Concluding remarks

This paper presented an outline and some useful properties of the LFTHREADS library and the responsibility hand-off (RHO) method, for which details can be found in [7, 8], along with a presentation of an implementation of LFTHREADS. The latter constitutes a proof-of-concept

[a] The interface we present here was chosen for brevity and simplicity. Our actual implementation aims to provide a POSIX threads compliant (IEEE POSIX 1003.1c) interface.

[b] I.e. here lock-free synchronization guarantees deadlock-avoidance among the operations implemented in lock-free manner, but an *application* that uses objects with blocking semantics (e.g. mutex) of course needs to take care to avoid deadlocks due to *inappropriate use* of the blocking operations by its threads.

lock-free implementation of the blocking mutex introduced in the paper and serving as basis for an experimental study of its properties. The experimental study performed in [7], using a mutex-intensive microbenchmark, shows positive figures. Moreover, the implementation can also serve as basis for further development, for porting the library to other platforms and for experimenting with parallel applications such as the Spark98 matrix kernels or the SPLASH-2 suite.

Although our present implementation of LFTHREADS is entirely at the user-level, its algorithms are well suited for use also in a kernel - user-level divided setting. With our method a significant benefit would be that there is no need for spin-locks and/or disabling interrupts in either the user-level or the kernel-level part.

References

[1] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Trans. on Computer Systems*, pages 53–79, 1992.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing,. In *Proc. of the 35th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[4] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *Proc. of the 18th Euromicro Conf. on Real-Time Systems*, pages 75–84. IEEE Computer Society, 2006.

[5] M. J. Feeley, J. S. Chase, and E. D. Lazowska. User-level threads and interprocess communication. Technical Report TR-93-02-03, University of Washington, Department of Computer Science and Engineering, 1993.

[6] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proc. of the Ottawa Linux Symp.*, pages 479–494, 2002.

[7] A. Gidenstam and M. Papatriantafilou. LFthreads: A lock-free thread library. In *Proc. of the 11th Int. Conf. on Principles of Distributed Systems (OPODIS)*, pages 217 – 231. Springer, 2007.

[8] A. Gidenstam and M. Papatriantafilou. LFthreads: A lock-free thread library. Technical Report MPI-I-2007-1-003, Max-Planck-Institut für Informatik, Algorithms and Complexity, 2007.

[9] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Practical and efficient lock-free garbage collection based on reference counting. In *Proc. of the 8th Int. Symp. on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pages 202 – 207. IEEE Computer Society, 2005.

[10] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. In *Proc. of the 13th Annual European Symp. on Algorithms (ESA)*, pages 329 – 242. Springer Verlag, 2005.

[11] M. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[12] M. B. Greenwald. *Non-blocking synchronization and system design*. PhD thesis, Stanford University, 1999.

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

[14] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. on Computer Systems*, 23(2):146–196, 2005.

[15] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[16] P. Holman and J. H. Anderson. Locking under pfair scheduling. *ACM Trans. Computer Systems*, 24(2):140–174, 2006.

[17] P. Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proc. of the 12th Int. Symp. on Distributed Computing (DISC)*, pages 216–230. Springer Verlag, 1998.

[18] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Trans. Computer Systems*, 15(1):3–40, 1997.

[19] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *Proc. of the 19th Int. Conf. on Distributed Systems (DISC)*, pages 354–368. Springer, 2005.

[20] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[21] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, 1991.

[22] M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. of SIGPLAN 2004 Conf. on Programming Languages Design and Implementation*, ACM SIGPLAN Notices. ACM Press, 2004.

[23] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department, 1995.

[24] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. of the 16th annual ACM Symp. on Principles of Distributed Computing*, pages 219–228, 1997.

[25] H. Oguma and Y. Nakayama. A scheduling mechanism for lock-free operation of a lightweight process library for SMP computers. In *Proc. of the 8th Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 235–242, 2001.

[26] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 204–213. ACM Press, 1995.

[27] Multithreading in the solaris operating environment. Technical report, Sun Microsystems, 2002.

[28] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.

[29] H. Sundell and P. Tsigas. NOBLE: A non-blocking interprocess communication library. In *Proc. of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*. Springer Verlag, 2002.

[30] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In *Proc. of the ACM SIGMETRICS 2001/Performance 2001*, pages 320–321. ACM press, 2001.

[31] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 214–222. ACM, 1995.

[32] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel processors. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):180–198, 1991.

[33] K. M. Zuberi and K. G. Shin. An efficient semaphore implementation scheme for small-memory embedded systems. In *Proc. of the 3rd IEEE Real-Time Technology and Applications Symp. (RTAS)*, pages 25–37. IEEE, 1997.