LFTHREADS: A Lock-Free Thread Library

Anders Gidenstam[1] and Marina Papatriantafilou[2]

[1] Algorithms and Complexity, Max-Planck-Institut für Informatik,
66123 Saarbrcken, Germany
andersg@mpi-inf.mpg.de
[2] Computer Science and Engineering, Chalmers University of Technology,
SE-412 96 Göteborg, Sweden
ptrianta@cs.chalmers.se

Abstract. LFTHREADS is 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. This is why a lock-free multithreading library is important. To the best of our knowledge LFTHREADS is the first thread library that provides a lock-free implementation of blocking synchronization primitives for application threads. Lock-free implementation of objects with blocking semantics may sound like a contradicting goal. However, such objects have 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, processors are always able to do useful work. As a consequence, applications, too, can enjoy enhanced parallelism and fault-tolerance. The synchronization in LFTHREADS is achieved by a new method, which we call *responsibility hand-off* (RHO), that does not need any special kernel support.

Keywords: lock-free, multithreading, multiprocessors, multicores, synchronization, shared memory.

1 Introduction

Multiprogramming and threading allow the processor(s) to be shared efficiently by several sequential threads of control. This paper studies synchronization algorithms for realizing standard thread-library operations and objects (create, exit, yield and mutexes) based entirely on *lock-free* methods. Lock-freedom implies that no spin-locks or similar locking synchronization is used in the implementation of the operations/objects and guarantees that in a set of concurrent operations at least one of them makes progress when there is interference and thus operations eventually completes.

The rationale in LFTHREADS is that processors should always be able to do useful work when there are runnable threads available, regardless of what other processors do;

i.e. despite other processors simultaneously accessing shared objects related with the implementation of the LFTHREADS-library operations 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 in this paper. 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. This is achieved via a new synchronization method, which we call *responsibility hand-off* (RHO), which may also be useful in lock-free synchronization constructions in general. 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 (ii) activating a blocked process 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 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 proposed and studied [1,2], 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 LFTHREADS's goal. 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 [3] and Sun Solaris [4]. 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 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. [5,6]) or the use of some application-related information (e.g. from real-time systems) to recover from it [7]; [8] and subsequent results inspired by it focus on scheduling with work-stealing, as a method to keep processors busy by providing fast and concurrent access to the set of ready threads; [9] aims at a similar direction, proposing thread scheduling that does not require locking (essentially using lock-free queuing) in a multithreading library called Lesser Bear; [10] studied methods of scheduling to reduce the amount of spinning in multithreaded mutual exclusion; [11] focuses on demands in real-time and embedded systems and studies methods for efficient, low-overhead semaphores; [12] gives an insightful overview of recent methods for mutual exclusion.

There has been other work at the operating system kernel level [13,14,15,16], 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. [17,18,19,20,21,22]).

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 [23] or software transactional memory constructions (cf. e.g. [24,25]).

2 Preliminaries

System model. The system consists of a set of processors, each having its own local memory as well as being connected to a shared memory through an interconnect network. Each processor executes instructions sequentially at an arbitrary rate. The shared memory might not be uniform, that is, for each processor the latency to access some part of the memory is not necessarily the same as the latency for any other processor to access that part. 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) used in the algorithms in this paper. These primitives are either available or can easily be derived from other available primitives [26,27] on contemporary microprocessor architectures.

Lock-free synchronization. *Lock-freedom* [28] 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. Other types of non-blocking synchronization are wait-freedom and obstruction-freedom. The correctness condition for atomic non-blocking operations is *linearizability* [29]. An execution is *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect at an atomic time instant that lies within its respective time duration, such that the effect of each operation is consistent with the effect of its corresponding operation in a sequential execution in which the operations appear in the same order.

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 [30,31]. Wait-free algorithms, as they provide stronger progress guarantees, are inherently more complex and more expensive than lock-free ones. Obstruction freedom implies weak progress guarantees and can be used e.g. for reference purposes, for studying parallelization.

In LFTHREADS the focus is on *lock-free synchronization* due to its combined benefits in progress, fault-tolerance and efficiency potential.

The problem and LFTHREADS's API. The LFTHREADS library defines the following procedures for thread handling[1]:
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 as well;
yield: causes the calling thread to be put on the ready queue and the (virtual) processor that 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 Detailed Description of the LFTHREADS Library

3.1 Data Structures and Fundamental Operations

We assume a data type, context_t, that can store the CPU context of an execution (i.e. thread) and some operations to manipulate such contexts (cf. Fig. 1). These operations, available in many operating systems[2], are:
(i) *save*(ctx) stores the state of the current CPU context in the supplied variable and switches the processor to a special system context. There is one such context for each processor. The return value from *save* is true when the context is stored and false when the context is restored.

[1] 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.

[2] In systems supporting the Single Unix Specification v2 (*SUSv2*), e.g. GNU/Linux, `getcontext(2)`, `setcontext(2)` and `makecontext(3)` can be used; in other Unix systems `setjump(3)` and `longjmp(3)` or similar.

```
type context_t is record ⟨implementation defined⟩;
function save(ctx : out context_t): boolean;
/* Saves the current CPU context and switches to a
 * system context. The call returns true when
 * the context is saved; false when it is restored. */
procedure restore(ctx : in context_t);
/* Replaces the current CPU context with a
 * previously stored CPU context.
 * The current context is destroyed. */
procedure make_context(ctx : out context_t;
        main : in pointer to procedure);
/* Creates a new CPU context which will wakeup
 * in a call to the procedure main when restored. */

type thread_t is record
    uc : context_t;

type lf_queue_t is record ⟨implementation defined⟩;
procedure enqueue(q : in out lf_queue_t;
        thread : in pointer to thread_t);
/* Appends the TCB thread to q. */
function dequeue(q : in out lf_queue_t;
        thread : out pointer to thread_t): boolean;
/* If the queue is not empty the first thread_t pointer
 * in the queue is dequeued and true is returned.
 * Returns false if the queue is empty. */
function is_empty(q : in out lf_queue_t): boolean;
/* Returns true if q is empty, false otherwise. */

function get_cpu_id(): cpu_id_t
/* Returns the ID of the current CPU (an int). */
```

```
/* Global shared variables. */
Ready_Queue : lf_queue_t;

/* Private per-processor persistent
 * variables. */
Current_p : pointer to thread_t;

/* Local temporary variables. */
next : pointer to thread_t;
old_count : integer;
old : cpu_id_t;

procedure create(thread : out thread_t;
        main : in pointer to procedure)
C1  make_context(thread.uc, main);
C2  enqueue(Ready_Queue, thread);

procedure yield()
Y1  if not is_empty(Ready_Queue) then
Y2      if save(Current_p.uc) then
Y3          enqueue(Ready_Queue, Current_p);
Y4          cpu_schedule();

procedure exit()
E1  cpu_schedule();

procedure cpu_schedule()
CI1 loop
CI2     if dequeue(Ready_Queue, Current_p)
        then
CI3         restore(Current_p.uc);
```

Fig. 1. The basic thread operations and shared data in LFTHREADS

(ii) *restore*(ctx) loads the supplied stored CPU context onto the processor. The restored context resumes execution in the (old) call to *save*, returning false. The CPU context that made the call to *restore* is lost (unless it was saved before).
(iii) *make_context*(ctx,main) creates a new CPU context. The new context starts in a call to the procedure main when it is loaded onto a processor with *restore*.

Each thread in the system will be represented by a thread control block (TCB) of type thread_t, containing a context_t field for storing the thread's state when it is not being executed on one of the processors.

Further, we assume we have a lock-free queue data structure (like e.g. [32]) for pointers to thread control blocks; the queue supports three lock-free and linearizable operations: *enqueue*, *dequeue* and *is_empty* (each with its intuitive semantics). The lock-free queue data structure is used as a building block in the implementation of LFTHREADS. However, as we will see in detail below, additional synchronization methods are needed to make operations involving more than one queue instance lock-free and linearizable.

3.2 Thread Operations in LFTHREADS

The general thread operations and variables used are shown in Fig. 1. The variables consist of the global shared Ready_Queue[3], which contains all runnable threads not

[3] The Ready_Queue here is a lock-free queue, but e.g. work-stealing [8] could be used.

currently being executed by any processor, and the per-processor persistent variable Current, which contains a pointer to the TCB of the thread currently being executed on that processor.

In addition to the public thread operations *create*, *exit* and *yield*, introduced above, there is an internal operation, *cpu_schedule*, used for selecting the next thread to load onto the processor. If there are no threads available in the Ready_Queue, the processor is idle and waits for a runnable thread to appear.

3.3 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.[4]

Mutex operations in LFTHREADS. The mutex_t structure (cf. Fig. 2) consists of three fields: (i) an integer counter, which counts the number of threads that are in or want to enter the critical section protected by the mutex; (ii) a lock-free queue, where the TCBs of blocked threads that want to lock the mutex is stored; and (iii) a hand-off flag, whose role and use will be described in detail below.

The operations on the mutex_t structure are shown in Fig. 2. In rough terms, 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 a new method, which

[4] 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.

we call the *responsibility hand-off* (RHO) method. 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. The details of the *responsibility hand-off* method are given in the description of the operations, below:

The lock operation: Line L1 atomically increases the count of threads that want to access the mutex using Fetch-And-Add. If the old value was 0 the mutex was free and is now locked by the thread. Otherwise the mutex is likely to be locked and the current thread has to block. Line L3 stores the context of the current thread in its TCB and line L4 enqueues the TCB on the mutex's waiting queue. From now on, this invocation of *lock* is not associated with any thread.

However, the processor cannot just leave and do something else yet, because the thread that owned the mutex might have unlocked it (since line L1); this is checked by line L6 to L8. If the token read from m.hand-off is not null then an *unlock* has tried to unlock the mutex but found (line U2) that although there is a thread waiting to lock the mutex, it has not yet appeared in the waiting queue (line H2). Therefore, the *unlock* has set the hand-off flag (line H5). However, it is possible that the hand-off flag was set after the thread enqueued by this *lock* (at line L4) had been serviced. Therefore, this processor should only attempt to take responsibility of the mutex if there is a thread available in the waiting queue. This is ensured by the *is_empty* test at line L7 and the CAS at line L8 which only succeeds if no other processor has taken responsibility of the mutex since line L6. If the CAS at line L8 succeeds, *lock* is now responsible for the mutex again and must find the thread wanting to lock the mutex. That thread (it might not be the same as the one enqueued by this *lock*) is dequeued from the waiting queue and this processor will proceed to execute it (line L9-L10). If the conditions at line L7 are not met or the CAS at line L8 is unsuccessful, the mutex is busy and the processor can safely leave to do other work (line L11).

To avoid ABA-problems (i.e. cases where CAS succeeds because the variable has been modified from its old value A to some value B and back to A) m.hand-off should, in addition to the processor id, include a per-processor sequence number. This is a well-known method in the literature, easy to implement and has been excluded from the presented code to make the presentation clearer.

The trylock operation: The operation will lock the mutex and return true if the mutex was unlocked. Otherwise it does nothing and returns false. The operation tries to lock the mutex by increasing the waiting count on line TL1. This will only succeed if the mutex was unlocked and there were no ongoing *lock* operations. If there are ongoing *lock* operations or some thread has locked the mutex, *trylock* will attempt to acquire the hand-off flag. If the *trylock* operation succeeds in acquiring the hand-off flag it becomes the owner of the mutex and increases the waiting count at line TL3 before returning true. Otherwise *trylock* returns false.

The *unlock* *operation:* If there are no waiting threads *unlock* unlocks the mutex. Otherwise one of the waiting threads is made owner of the mutex and enqueued on the Ready_Queue. The operation begins by decreasing the waiting count at line U1, which was increased by this thread's call to *lock* or *trylock*. If the count becomes 0, there are no waiting threads and the *unlock* operation is done. Otherwise, there are at least one thread wanting to acquire the mutex and the *do_hand-off* procedure is used to either find the thread or hand-off the responsibility for the mutex. If the waiting thread has been enqueued in the waiting queue, it is dequeued (line H2) and moved to the Ready_Queue (line H3) which completes the *unlock* operation. Otherwise, a *responsibility hand-off* is initiated to get rid of the responsibility for the mutex (line H5):

- The responsibility hand-off is successful and terminates if: (i) the waiting queue is still empty at line H6; in that case either the offending thread has not yet been enqueued there (in which case, it has not yet checked for hand-offs) or it has in fact already been dequeued (in which case, some other processor took responsibility for the mutex); or if (ii) the attempt to retake the hand-off flag at line H8 fails, in which case, some other processor has taken responsibility for the mutex. After a successful hand-off the processor leaves the *unlock* procedure (line H7 and H9).
- If the hand-off is unsuccessful, i.e. the CAS at line H8 succeeds, the processor is again responsible for the mutex and must repeat the hand-off procedure. Note that when a hand-off is unsuccessful, at least one other concurrent *lock* operation made progress, namely by completing an enqueue on the waiting queue (otherwise this *unlock* would have completed at lines H6-H7). Note further that since the CAS at line H8 succeeded, none of the concurrent *lock* operations have executed line L6-L8 since the hand-off began.

Fault-tolerance. Regarding *processor failures*, the procedures enable the highest achievable level of fault-tolerance for a mutex. Note that even though a *processor failure* while the *unlock* is moving a thread from the m.waiting queue to the Ready_Queue (between line H2 and H3) could cause the loss of two threads (i.e. the current one and the one being moved), the system behaviour in this case is indistinguishable from the case when the processor fails before line H2. In both cases the thread owning the mutex has failed before releasing ownership. At all other points a *processor failure* can cause the loss of at most one thread.

4 Correctness of the Synchronization in LFTHREADS

To prove the correctness of the thread library we need to show that the mutex primitive has the desired semantics. We will first show that the mutex operations are lock-free and linearizable with respect to the processors and then that the lock-free mutex implementation satisfies the conditions for mutual exclusion with respect to the application threads. First we (i) define some notation that will facilitate the presentation of the arguments and (ii) establish some lemmas that will be used later to prove the safety,

```
type mutex_t is record
       waiting : lf_queue_t;
       count : integer := 0;
       hand-off : cpu_id_t := null;

procedure lock(m : in out mutex_t)
L1   old_count := FAA(&m.count, 1);
L2   if old_count $\neq 0$ then
          /* The mutex was locked.
           * Help or run another thread. */
L3      if save(Current$_p$.uc) then
L4         enqueue(m.waiting, Current$_p$);
L5         Current$_p$ := null;
           /* The thread is now blocked. */
L6         old := m.hand-off;
L7         if old $\neq$ null and
             not is_empty(m.waiting) then
L8            if CAS(&m.hand-off, old, null)
              then /* We now own m; */
                 /* ... run a blocked thread */
L9               dequeue(m.waiting, Current$_p$);
L10              restore(Current$_p$); /* Done. */
L11        cpu_schedule(); /* Done. */

function trylock(m : in out mutex_t): boolean
TL1 if CAS(&m.count, 0, 1) then return true;
TL2 else if GrabToken(&m.hand-off) then
TL3      FAA(&m.count, 1);
TL4      return true;
TL5 return false;
```

```
procedure unlock(m : in out mutex_t)
U1   old_count := FAA(&m.count, $-1$);
U2   if old_count $\neq 1$ then
         /* There is a waiting thread. */
U3       do_hand-off(m);

procedure do_hand-off(m : in out mutex_t)
H1   loop /* We own the mutex. */
H2      if dequeue(m.waiting, next) then
H3         enqueue(Ready_Queue, next);
H4         return; /* Done. */
        else
           /* The waiting thread isn't ready! */
H5         m.hand-off := get_cpu_id();
H6         if is_empty(m.waiting) then
              /* Some concurrent operation will
               * see/or has seen the hand-off. */
H7            return; /* Done. */
H8         if not CAS(&m.hand-off,
                    get_cpu_id(), null) then
              /* Some concurrent operation
               * acquired the mutex. */
H9            return; /* Done. */

function GrabToken(loc : pointer to cpu_id_t)
  : boolean
GT1 old := *loc;
GT2 if old = null then return false;
GT3 return CAS(loc, old, null);
```

Fig. 2. The lock-free mutex protocol in LFTHREADS

liveness, fairness and atomicity properties of the algorithm. Due to space constraints the full proofs can be found in [33].

Definition 1. *A thread's call to a blocking operation Op is said to be* completed *when the processor executing the call leaves the blocked thread and goes on to do something else (e.g. executing another thread). The call is said to have* returned *when the thread (after becoming unblocked) continues its execution from the point of the call to Op.*

Definition 2. *A mutex* m *is* locked *when* $\mathsf{m.count} > 0$ *and* $\mathsf{m.hand\text{-}off} = null$*. Otherwise it is* unlocked*.*

Definition 3. *When a thread τ's call to* lock *on a mutex* m *returns we say that thread τ has* locked *or* acquired *the mutex* m*. Similarly, we say that thread τ has* locked *or* acquired *the mutex* m *when the thread's call to* trylock *on the mutex* m *returns $True$. Further, when a thread τ has acquired a mutex* m *by a* lock *or successful* trylock *operation and not yet released it by calling* unlock *we say that the thread τ is the* owner *of the mutex* m *(or that τ* owns m*).*

Lock-freedom. The lock-free property of the thread library operations will be established with respect to the processors. An operation is lock-free if it is guaranteed to complete in a bounded number of steps unless it is interfered with an unbounded number of times by other operations and every time operations interfere, at least one of them is guaranteed to make progress towards completion.

Theorem 1. *The mutex operations lock, trylock and unlock are all lock-free.*

The lock-freedom of *trylock* and *unlock* with respect to application threads follows from their lock-freedom with respect to the processors, as they do not contain context switches. The operation *lock* is neither non-blocking nor lock-free for application threads, since a call to *lock* on a locked mutex should block.

Linearizability. Linearizability guarantees that the result of any concurrent execution of operations is identical to a sequential execution where each operation takes effect atomically at a single point in time (its *linearization point*) within its duration in the original concurrent execution.

Theorem 2. *The mutex operations lock, trylock and unlock are linearizable.*

Mutual exclusion properties. The mutual exclusion properties of the new mutex protocol are established with respect to application threads.

Theorem 3 (Safety). *For any mutex m and at any time t there is at most one thread τ such that τ is the owner of m at time t.*

Theorem 4 (Liveness I). *A thread τ waiting to acquire a mutex m eventually acquires the mutex once its lock operation has enqueued τ on the m.waiting queue.*

Theorem 5 (Liveness II). *A thread τ wanting to acquire a mutex m can only be starved if there is an unbounded number of lock operations on m performed by threads on other processors.*

Theorem 6 (Fairness). *A thread τ wanting to acquire a mutex m only has to wait for the threads enqueued on the m.waiting queue before τ was enqueued.*

5 Experimental Study

The primary contribution of this work is to enhance qualitative properties of thread library operations, such as the tolerance to delays and processor failures. However, since lock-freedom may also imply performance/scalability benefits with increasing number of processors, we also wanted to observe this aspect. We made an implementation of the mutex object and the thread operations on the GNU/Linux operating system. The implementation is written in the C programming language and was done entirely at the user-level using "cloned"[5] processes as *virtual processors* for running the threads. The implementation uses the lock-free queue in [32] for the mutex waiting queue and the Ready_Queue. To ensure sufficient memory consistency for synchronization variables, memory barriers surround all CAS and FAA instructions and the writes at lines L6 and H5. The lock-based mutex implementation uses a test and test-and-set spin-lock

[5] "Cloned" processes share the same address space, file descriptor table and signal handlers etc and are also the basis of Linux's native pthread implementation.

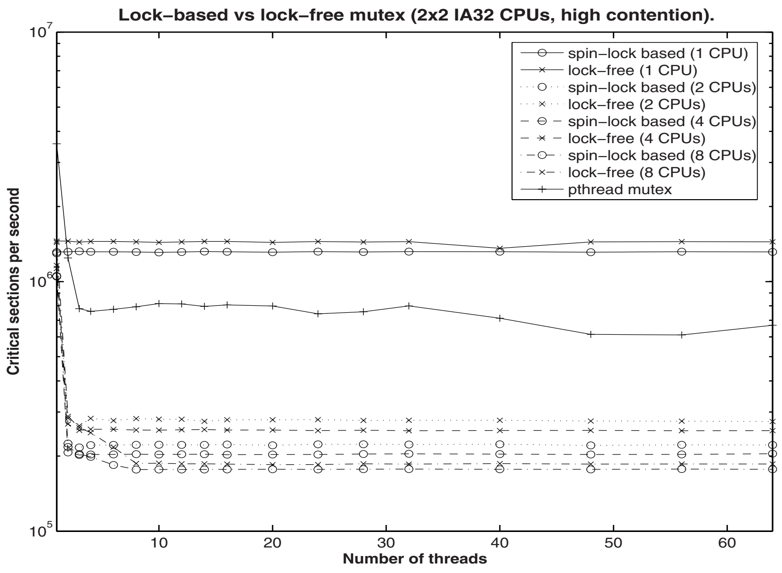

Fig. 3. Mutex performance in LFTHREADS and pthreads at high contention

to protect the mutex state. Unlike the use of spin-locks in an OS kernel, where usually neither preemptions nor interrupts are allowed while holding a spin-lock, our virtual processors can be interrupted by the OS kernel due to such events. This behaviour matches the asynchronous processors in our system model.

The experiments were run on a PC with two Intel Xeon 2.80GHz processors (acting as 4 due to hyper-threading) using the GNU/Linux operating system with kernel version 2.6.9. The microbenchmark used for the experimental evaluation consists of a single critical section protected by a mutex and a set of threads that each try to enter the critical section a fixed number of times. The contention level on the mutex was controlled by changing the amount of work done outside the critical section. We evaluated the following configurations experimentally:

- The lock-free mutex using the protocol presented in this paper, using 1, 2, 4 and 8 virtual processors to run the threads.
- The spin-lock based mutex, using 1, 2, 4 and 8 virtual processors.
- The platform's standard pthreads library and a standard pthread mutex. The pthreads library on GNU/Linux use kernel-level "cloned" processes as threads, which are scheduled on all available processors, i.e. the pthreads are at the same level as the virtual processors in LFTHREADS. The difference in scheduling makes it difficult to interpret the pthreads results with respect to the others; i.e. the pthreads results are primarily for reference.

Each configuration was run 10 times; the diagrams present the mean.

High contention. Fig. 3 shows the results when no work is done outside the critical section, i.e. the contention on the mutex is high. The desired result here is that throughput for an implementation stays the same regardless of the number of threads or (virtual) processors. This would imply that the synchronization scales well. However, in reality the throughput decreases with increasing number of virtual processors, mainly due to preemptions inside the critical section (but for spin-locks also inside mutex operations) and synchronization overhead. The results indicate that the lock-free mutex has less overhead than the lock-based.

Low contention. Fig. 4 shows the results when the threads perform 1000 times more work outside the critical section than inside, making the contention on the mutex low. With the majority of the work outside the critical section, the expected behaviour is a linear throughput increase over threads until all (physical) processors are in use by threads, thereafter constant throughput as the processors are saturated with threads running outside the critical section. The results agrees with the expected behaviour; we see that from one to two virtual processors the throughput doubles in both the lock-free and spin-lock based cases. (Recall that the latter is a test-and-test-and-set-based implementation, which is favoured under low contention). Note that the step to 4 virtual processors does not double the throughput — this is due to hyper-threading, there are not 4 physical processors available. Similar behaviour can also be seen in the pthread-based case. The lock-free mutex shows similar or higher throughput than the spin-lock-based for the same number of virtual processors; it also shows comparable and even better performance than the pthread-based when the number of threads is large and there are more virtual processors than physical.

Summarizing, we observe that LFTHREADS's lock-free mutex protocol implies comparable or better throughput than the lock-(test-and-test-and-set-)based implementation, both in high- and in low-contention scenaria for the same number of virtual processors,

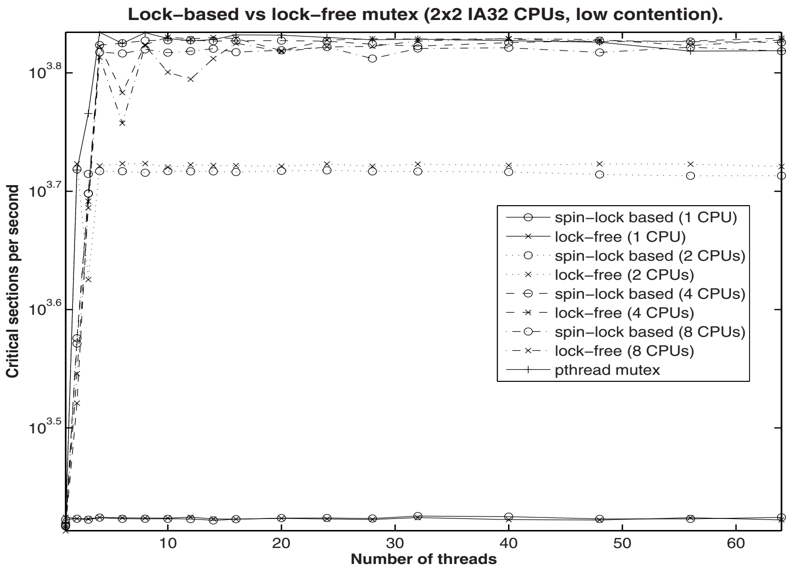

Fig. 4. Mutex performance in LFTHREADS and pthreads at low contention

besides offering the qualitative advantages in tolerance against slow, delayed or crashed threads, as discussed earlier in the paper.

6 Conclusion

This paper presented the LFTHREADS library and the responsibility hand-off (RHO) method. Besides supporting a thread-library interface with lock-free implementation of a blocking synchronization primitive and fault-tolerance properties, the RHO method can be regarded as a conceptual contribution, which can be useful in lock-free synchronization in general.

The present implementation of LFTHREADS is done entirely at the user-level, but the algorithms are well suited for use also in a kernel - user-level divided setting. A significant benefit of the new method there is that neither modifications to the operating system kernel nor spin-locks and/or disabling of interrupts are needed in the user-level or the kernel-level part. LFTHREADS constitutes a proof-of-concept of lock-free implementation of the blocking mutex introduced in the paper and serves as basis for an experimental study of its performance. The experimental study performed here, 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 multiprocessors and experimenting with parallel applications such as the Spark98 matrix kernels or the SPLASH-2 suite.

Acknowledgements. We thank the anonymous reviewers for useful comments.

References

1. Anderson, T., Bershad, B., Lazowska, E., Levy, H.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In: ACM Trans. on Computer Systems, pp. 53–79. ACM Press, New York (1992)
2. Feeley, M.J., Chase, J.S., Lazowska, E.D.: User-level threads and interprocess communication. Technical Report TR-93-02-03, University of Washington, Department of Computer Science and Engineering (1993)
3. Franke, H., Russell, R., Kirkwood, M.: Fuss, futexes and furwocks: Fast userlevel locking in linux. In: Proc. of the Ottawa Linux Symp, pp. 479–494 (2002)
4. Multithreading in the solaris operating environment. Technical report, Sun Microsystems
5. Kontothanassis, L.I., Wisniewski, R.W., Scott, M.L.: Scheduler-conscious synchronization. ACM Trans. Computer Systems 15(1), 3–40 (1997), doi:10.1145/244764.244765
6. Holman, P., Anderson, J.H.: Locking under pfair scheduling. ACM Trans. Computer Systems 24(2), 140–174 (2006)
7. Devi, U.C., Leontyev, H., Anderson, J.H.: Efficient synchronization under global edf scheduling on multiprocessors. In: Proc. of the 18th Euromicro Conf. on Real-Time Systems, pp. 75–84. IEEE Computer Society, Los Alamitos (2006)
8. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing, In: Proc. of the 35th Annual Symp. on Foundations of Computer Science (FOCS), 356–368 (1994)

9. Oguma, H., Nakayama, Y.: 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), 235–242 (2001)
10. Zahorjan, J., Lazowska, E.D., Eager, D.L.: 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)
11. Zuberi, K.M., Shin, K.G.: An efficient semaphore implementation scheme for small-memory embedded systems. In: Proc. of the 3rd IEEE Real-Time Technology and Applications Symp (RTAS), IEEE, pp. 25–37. IEEE Computer Society Press, Los Alamitos (1997)
12. Anderson, J.H., Kim, Y.J., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Distributed Computing 16(2-3), 75–110 (2003)
13. Massalin, H., Pu, C.: A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91 (1991)
14. Massalin, H.: Synthesis: An Efficient Implementation of Fundamental Operating System Services. PhD thesis, Columbia University (1992)
15. Greenwald, M., Cheriton, D.R.: The synergy between non-blocking synchronization and operating system structure. In: Operating Systems Design and Implementation, 123–136 (1996)
16. Greenwald, M.B.: Non-blocking synchronization and system design. PhD thesis, Stanford University (1999)
17. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC), ACM, pp. 214–222. ACM Press, New York (1995)
18. Michael, M.M., Scott, M.L.: Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department (1995)
19. Michael, M.: Scalable lock-free dynamic memory allocation. In: Proc. of SIGPLAN 2004 Conf. on Programming Languages Design and Implementation, ACM Press, ACM SIGPLAN Notices (2004)
20. Gidenstam, A., Papatriantafilou, M., Sundell, H., Tsigas, P.: 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), pp. 202–207. IEEE Computer Society Press, Los Alamitos (2005)
21. Gidenstam, A., Papatriantafilou, M., Tsigas, P.: Allocating memory in a lock-free manner. In: Proc. of the 13th Annual European Symp. on Algorithms (ESA), pp. 242–329. Springer, Heidelberg (2005)
22. Herlihy, M., Luchangco, V., Martin, P., Moir, M.: Nonblocking memory management support for dynamic-sized data structures. ACM Trans. on Computer Systems 23(2), 146–196 (2005)
23. Sundell, H., Tsigas, P.: NOBLE: A non-blocking inter-process communication library. In: Sundell, H., Tsigas, P. (eds.) Proc. of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, Springer, Heidelberg (2002)
24. Marathe, V.J.I.W.N.S., Scott, M.L: Adaptive software transactional memory. In: Proc. of the 19th Int. Conf. on Distributed Systems (DISC), Springer, pp. 354–368. Springer, Heidelberg (2005)
25. Shavit, N., Touitou, D.: Software transactional memory. In: Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC), pp. 204–213. ACM Press, New York (1995)
26. Jayanti, P.: 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), pp. 216–230. Springer, Heidelberg (1998)

27. Moir, M.: Practical implementations of non-blocking synchronization primitives. In: Proc. of the 16th annual ACM Symp. on Principles of Distributed Computing, pp. 219–228. ACM Press, New York (1997), citeseer.ist.psu.edu/moir97practical.html
28. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Trans. on Programming Languages and Systems 15(5), 745–770 (1993)
29. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. on Programming Languages and Systems 12(3), 463–492 (1990), http://www.acm.org/pubs/toc/Abstracts/0164-0925/78972.html
30. Sundell, H.: Efficient and Practical Non-Blocking Data Structures. PhD thesis, Chalmers University of Technology (2004)
31. Tsigas, P., Zhang, Y.: Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In: Proc. of the ACM SIGMETRICS 2001/Performance 2001, pp. 320–321. ACM Press, New York (2001)
32. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: Proc. 13th ACM Symp. on Parallel Algorithms and Architectures, pp. 134–143. ACM Press, New York (2001)
33. Gidenstam, A., Papatriantafilou, M.: LFthreads: A lock-free thread library. Technical Report MPI-I-2007-1-003, Max-Planck-Institut für Informatik, Algorithms and Complexity (2007)