“Blocking without Locking” or LFTHREADS: A lock-free thread library

Anders Gidenstam
Marina Papatriantafilou

MPI–I–2007–1–003 October 2007

Authors' Addresses

Anders Gidenstam
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany

Marina Papatriantafilou
Computer Science and Engineering
Chalmers University of Technology
SE 412 96 Göteborg
Sweden.

Abstract

This paper presents the synchronization in 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. 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.

Contents

1 Introduction

Multiprogramming and threading allow the processor(s) to be shared by several sequential threads of control efficiently. Although they are a fundamental part of modern operating systems, their implementation are all the more important in the context of multiprocessor or multicore systems. In this paper we study synchronization algorithms for realizing standard thread-library operations and objects (create, exit, yield and mutexes), in a thread library LFTHREADS, based entirely on *lock-free* methods. Lock-freedom implies that no spin-locks or similar locking synchronization methods are used in the implementation of the operations/objects and guarantees that in a set of concurrent operations at least one of them makes progress every time there is interference and thus operations eventually complete.

The rationale in LFTHREADS is that processors should always be able to do useful work when there are runnable threads available (i.e. not blocked by the application), regardless of what other processors do; i.e. a processor is able to continue doing useful work, 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).

Note that 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 allows such application threads to be blocked *without enforcing mutual exclusion among the processors* executing the threads. This means that 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*. 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 com-

plete. To achieve this, we introduce a new synchronization method, which we call *responsibility hand-off* (RHO), which may also be useful in other lock-free synchronization constructions. 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 employ the use of 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 [2, 8], 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 also observed that application-controlled blocking and interprocess communication can be resolved at the 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 the threads with 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 [9] and Sun Solaris [33]. 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 way that is non-blocking to the threads and processors 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 done entirely at the user-level, the algorithms used in it 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 over the kernel/scheduler to avoid unwanted preemption (cf. e.g. [23, 21]) or the use of application-related information (e.g. period and execution time bounds in real-time systems) to recover from it [7]; [4] 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; [31] 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. [40] studied methods of scheduling to reduce the amount of spinning in multithreaded mutual exclusion; [41] 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 [27, 26, 13, 14], 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, memory allocation, e.g. [39, 29, 6, 28, 11, 12, 17]). In [5], the topic of lock-free synchronization in multithreading libraries is addressed, too: it presents a lock-free algorithm to replace mutually exclusive access to multi-word objects, where the access can consist of any side-effect-free function producing the new object state.

The goal of LFTHREADS is to implement a common 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 e.g. the NOBLE library [35] that provides implementations of a large range of data structures using lock-free methods, or other constructions that aim to provide support for non-blocking programming, e.g. the Software Transactional Memory package for C# [16] and the many non-blocking STM algorithms in the literature, e.g. [19, 25, 32, 10].

The paper is organized as follows: first we present the system model together with some background information on lock-free synchronization and the problem we focus on including the application programming interface of LFTHREADS (Chapter 2); followed by a detailed description of the algorithmic design (Chapter 3); the correctness of the above (Chapter 4); some implementation-related information and an experimental study (Chapter 5). We conclude in Chapter 6.

2 Preliminaries

System model

We consider shared memory multiprocessor systems, where 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 of the shared memory. 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) (see Figure 1) used in the algorithms in this paper. These primitives are either available or can easily be derived from other available primitives [22, 30] on most contemporary microprocessor architectures. The processors in the system cooperate to run a set of application threads. Each thread consists of a sequence of operations; communication is accomplished via shared-memory operations.

Lock-free synchronization

Lock-freedom [15] is a type of non-blocking synchronization that guarantees that in a set of concurrent operations at least one of them makes progress and thus eventually completes each time. Another type of non-blocking synchronization is *wait-freedom* [24], which guarantees that *every* operation finishes in a finite number of its own steps regardless of the actions of concurrent operations. In the literature we also see *obstruction-freedom* [18], a weak non-blocking synchronization option, guaranteeing only that, at any point, a thread that executes *alone* for a sufficiently large but bounded number of steps can complete its operation. Obstruction free algorithms are distinguished from lock-free and wait-free ones: in the former, progress is not guaranteed in presence of concurrency and operations may even abort.

Figure 1 The Compare-And-Swap (CAS) and Fetch-And-Add (FAA) atomic primitives.

```
function CAS(address : pointer to word;
        oldvalue : word; newvalue : word) : boolean
    atomic do
        if *address = oldvalue then
            *address := newvalue;
            return true;
        else return false;
```

```
function FAA(address: pointer to integer;
        increment: integer): integer
    atomic do
        ret := *address;
        *address := ret + increment;
        return ret;
```

The correctness condition for atomic non-blocking operations is *linearizability* [20]. 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 a number of advantages over lock-based synchronization: (i) it does not give rise to priority inversion; (ii) it avoids lock convoys; (iii) it provides better fault tolerance (processor stop failures will not corrupt shared data objects); and (iv) it eliminates deadlock scenarios involving two or more threads both waiting for each other. Due to these facts there is extended research literature on lock-free synchronization (c.f. [34] for an overview) as well as on *universal methods* to transform lock-based constructions into lock-free/wait-free ones (e.g. [3, 15, 38]). Besides ensuring the above qualitative properties, 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 [34, 36]. Wait-free algorithms, as they provide stronger progress guarantees, are inherently more complex and more expensive than lock-free ones. Obstruction freedom implies very 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]:
procedure create(thread : **out** thread_t; main : **in pointer to procedure**);
procedure exit();

[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.

procedure yield();

Procedure *create* creates a new thread which will start in the procedure main. Procedure *exit* terminates the calling thread and if this was the last thread of the application/process the latter is terminated as well. Procedure *yield* causes the calling thread to be put on the ready queue and the (virtual) processor that was running it to pick a new thread to run from the ready queue.

For applications that need lock-based synchronization between threads the thread library provides a mutex object. The mutex object supports the operations:

procedure lock(mutex : **in out** mutex_t)

procedure unlock(mutex : **in out** mutex_t)

function trylock(mutex : **in out** mutex_t): **boolean**

Procedure *lock* attempts to lock the mutex. If the mutex is locked already the calling thread is blocked and enqueued on the waiting queue of the mutex. Procedure *unlock* unlocks a mutex if there are no threads waiting in the mutex's waiting queue, otherwise the first of the waiting threads are removed from the waiting queue and made runnable. That thread becomes the new owner of the mutex. Only the thread owning the mutex may call *unlock*. Function *trylock* tries to lock the mutex. If it succeeds (i.e. the mutex was unlocked) true is returned, otherwise false.

3 Detailed description of the LFTHREADS library

3.1 Data structures used in LFTHREADS

In Figure 2 the data structures used in the implementation of the LFTHREADS library are presented. We assume that we have a data type, context_t, where the CPU context of an execution (i.e. thread) can be stored and some operations to manipulate such contexts. These operations, which can be supported by most common operating systems[1], are:
(i) *save*(context) stores the state of the current CPU context in the supplied variable and switches processor to a special system context. There is one such context available for each processor. The return value from *save* is true when the context is stored and false when the context is restored.
(ii) *restore*(context) 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 the call to *restore*).
(iii) *make_context*(context,main) creates a new CPU context in the supplied variable. The new context will start 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 an instance of the thread control block data type, thread_t, which contains a context_t variable that stores the thread's state when it is not being executed on one of the processors.

Further, we also assume that we have a lock-free queue data structure (like e.g. [37]) for pointers to thread control blocks; the queue supports three lock-free

[1] For example in systems conforming to the Single Unix Specification v2 (*SUSv2*), such as GNU/Linux, they can be implemented from `getcontext(2)`, `setcontext(2)` and `makecontext(3)`, while in other Unix systems `setjump(3)` and `longjmp(3)` or similar could be used.

Figure 2 Thread context and thread queue operations used in LFTHREADS.

```
type context_t is record ⟨implementation defined⟩;

function save(context : out context_t): boolean;
/* Saves the current CPU context and switches to a
 * system context. The call to save returns true when
 * the context is saved and false when it is restored. */
procedure restore(context : in context_t);
/* Replaces the current CPU context with a
 * previously stored CPU context.
 * The current context is destroyed. */
procedure make_context(context : 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;
/* Thread control block. */

type lf_queue_t is record ⟨implementation defined⟩;

procedure enqueue(queue : in out lf_queue_t;
          thread : in pointer to thread_t);
/* Appends the thread control block thread to
 * the end of the queue. */
function dequeue(queue : 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(queue : in out lf_queue_t): boolean;
/* Returns true if the queue is empty, and
 * false otherwise. */

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

Figure 3 The basic thread operations and shared data in LFTHREADS.

```
/* 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);
```

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 in LFTHREADS are shown in Figure 3. The persistent global and per-processor variables consist of the global shared Ready_Queue[2], which contains all runnable threads not currently being executed by any processor, and the per-processor persistent variable Current, which contains a pointer to the thread control block of the thread currently being executed on that processor.

The thread handling operations, whose required functionality was introduced in section 2, work as follows in LFTHREADS:
(i) Operation *create* creates a new thread control block, initializes the context stored in the block and enqueues the new thread on the ready queue.
(ii) Operation *exit* terminates the thread currently being executed by this processor, which then picks another thread to run from the ready queue.
(iii) Operation *yield* saves the context of the thread currently being executed by this processor, enqueues this thread on the ready queue and then picks another thread to run from the ready queue.

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

3.3 Blocking thread synchronization in LFTHREADS 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 individual application thread trying to lock a mutex will be blocked if the mutex has been locked by another application thread. A faulty application can also deadlock its threads. It is the responsibility of the application developer to prevent such

[2]The Ready_Queue here is a lock-free queue, but e.g. work-stealing [4] could be used.

situations.[3]

Mutex operations in LFTHREADS

The mutex_t structure, shown in Figure 4, 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 thread control blocks of blocked threads wanting to lock the mutex when it is already locked can be 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 Figure 4. 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 cause situations such that e.g. a thread A finds the mutex locked, but 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 at a time modifies the mutex state, i.e. by enforcing mutual exclusion among the processors in the implementation of the mutex operations, 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 we call the *responsibility hand-off* (RHO) method. In particular, the thread/processor that is 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 thread/processor.

By performing a *responsibility hand-off*, the processor executing the *unlock* operation can finish this operation and continue executing threads without needing to wait for any concurrent *lock* operations 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

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

Figure 4 The lock-free mutex protocol in LFTHREADS.

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

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

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

procedure unlock(mutex : in out mutex_t)
U1      old_count := FAA(&mutex.count, −1);
U2      if old_count ≠ 1 then
            /* There is at least one waiting thread. */
U3          do_hand-off(mutex);

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

useful work, including performing 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 (at 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 although 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 at 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. This might succeed if the thread owning the mutex is trying to unlock it and did not find any thread in the waiting queue despite at least one ongoing *lock* operation. 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 in order 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, the waiting queue is empty and the *unlock* operation initiates a *responsibility hand-off* 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, this processor is yet again responsible for the mutex and must repeat the hand-off procedure. Note that when a hand-off is unsuccessful, at least some 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, namely the one whose context is executing.

4 Correctness of the synchronization in LFTHREADS

To prove the correctness of the synchronization in 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 behaviour of the application threads.

First we define (i) some notation that will facilitate the presentation of the arguments and (ii) establish some lemmas that will be used later to prove the safety, liveness, fairness and atomicity properties of the algorithm.

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* m.count > 0 *and* m.hand-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*).*

Lemma 1 *The value of the* m.count *variable is never negative and always greater than zero when a thread* owns *the mutex* m.

Proof: Note that m.count is increased by one for each *lock* (line L1) and each successful *trylock* (line TL1 or TL3) operation and it is decreased by one for each *unlock* operation (line U1). Therefore, m.count cannot be zero unless the number of calls to *unlock* is the same as the number of calls to *lock* and successful calls to *trylock* together. In a correct application all threads must have a matching *unlock* call after each *lock* (or *trylock*) call and no *unlock* calls without a matching *lock* (or *trylock*). For a thread τ to own the mutex it must have called *lock* (or *trylock*) but not (yet) called *unlock* after that, so m.count must be positive. Additionally m.count cannot become negative since there can never be more calls to *unlock* than to *lock* (or successful ones to *trylock*) . □

Lemma 2 *If m.hand-off* $\neq null$ *then m.count* > 0.

Proof: There is only one place in the mutex code where m.hand-off is set to something $\neq null$, namely line H5 in *do_hand-off* which is called from *unlock*.

First observe that only a thread that owns the mutex is allowed to call *unlock* and therefore by Lemma 1 m.count > 0 when any *unlock* operation begins.

Let $\langle x, y\rangle$ denote the mutex state m.count $= x$, m.hand-off $= y$ and assume towards a contradiction that the state $\langle 0, \neq null\rangle$ has been reached.

Consider the *unlock*, call it A, that set m.hand-off $\neq null$. Note that when it executed line U1 m.count > 1 or else it would never enter *do_hand-off*. Therefore another *unlock*, B, must have decreased m.count afterwards. This *unlock* could only be executed by a thread τ that became owner of the mutex by a *lock* or *trylock* that took effect after A released the mutex. There are two cases for how τ could gain ownership: (i) the *unlock* A activated τ at line H3. This contradicts the assumption the A reached line H5.
(ii) The *lock* or *trylock* by τ grabbed the hand-off token set by A at line H5. Since the only possible way A can reset the hand-off token again requires that A successfully grabs its own hand-off token at line H8 this is also a contradiction. □

4.1 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.*

Proof: The only instance of non-sequential code is the hand-off loop in *do_hand-off* called by the operation *unlock*. The conditions that must hold for the processor to stay in the loop are:

(i) the m.waiting queue is empty at line H2; and
(ii) the m.waiting queue is non-empty at line H6; and
(iii) the processor successfully captures the m.hand-off flag at line H8.

For both (i) and (ii) to hold, at least one other processor must have completed an enqueue operation on the m.waiting queue between the execution of line H2 and H6 and thus have made progress.[1] □

The lock-freedom of *trylock* and *unlock*, with respect to application threads, follows trivially from their lock-freedom with respect to processors, since there are no context switches in them. The operation *lock* it is clearly neither non-blocking nor lock-free with respect to application threads, since a thread calling *lock* on a locked mutex should be blocked.

4.2 Linearizability

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

Theorem 2 *Operation lock is linearizable.*

Proof: A *lock* either can succeed to lock the mutex immediately without blocking or it has to block until the current owner of the mutex unlocks it. The LP of *lock* is when the thread becomes the owner of the mutex. Let $\langle x, y \rangle$ denote the mutex state m.count $= x$, m.hand-off $= y$. There are two main cases for a call to *lock* by a thread τ:

(I) The mutex is unlocked with no ongoing operations $\langle 0, null \rangle$. In this case the LP of the *lock* is the FAA instruction at line L1. It atomically changes the mutex state to locked $\langle 1, null \rangle$. The *lock* operation then returns to the calling thread τ.

(II) There are other ongoing operations on the mutex, indicated by m.count > 0 at line L1. In this case the thread τ is blocked (line L3) and enqueued on the m.waiting queue (line L4). The processor then continues the *lock* operation. There are two possible sub-cases:
(i) The CAS on line L8 is not reached or is unsuccessful. Then the thread

[1] Note that condition (iii) is irrelevant for the proof of lock-freedom.

τ will be activated (and thus given an LP) by either a concurrent or future *unlock* (see *unlock* below) or another concurrent *lock* that successfully passed line L8 (see sub-case (ii) below).
(ii) The CAS at line L8 succeeds. This implies that this processor is now responsible for the mutex and that there is a thread present in the *m.waiting* queue. The latter is guaranteed by the fact that threads can only be dequeued from *m.waiting* by a processor owning the mutex and that the ownership token cannot have changed between line L6 and L8 for the CAS to succeed. Then the LP of the *lock* operation of the thread τ' dequeued at line L9 (which may or may not be τ, see sub-case (i) above) is the LP of the *dequeue*.

□

Theorem 3 *Operation trylock is linearizable.*

Proof: A *trylock* can either lock the mutex in which case it returns $True$ or, if the mutex is already locked, do nothing and return $False$.

(I) The *trylock* by a thread τ returns $True$. In this case either the CAS at line TL1 or the CAS at GT3 (called from TL2) succeeded. It is easy to see that the successful CAS changes the mutex state from unlocked to locked atomically and forms the LP of *trylock*.

(II) The *trylock* by a thread τ returns $False$. In this case the LP is between the CAS at line TL1 and the CAS at GT3 (called from TL2). Observe that since *trylock* returns $False$ it must be the case that m.count $\neq 0$ at line LT1 and that m.hand-off $= null$ or m.hand-off $\neq$ old at line GT3. Let $\langle x, y\rangle$ denote m.count $= x$, m.hand-off $= y$. The cases are:
(i) m locked $\langle > 0, null\rangle$ at line TL1: then the LP is TL1;
(ii) m unlocked $\langle > 0, \neq null\rangle$ at line TL1 and m locked $\langle > 0, null\rangle$ at line GT3: then the LP is GT3;
(iii) m unlocked $\langle > 0, \neq null\rangle$ at line TL1 and m unlocked $\langle 0, null\rangle$ at line GT3: in this case it follows from Lemma 2 that some other thread τ' must have concurrently locked the mutex $\langle > 0, null\rangle$, and subsequently unlocked it when there were no waiting threads $\langle 0, null\rangle$. Hence the LP is in the interval between the *lock* and *unlock* of τ';
(iv) the CAS at GT3 fails due to m.hand-off having changed from one non-null value to another since line GT1: in this case the LP is as in case (iii), since the m.hand-off change implies that the mutex became locked and then unlocked again.

□

Theorem 4 *Operation unlock is linearizable.*

Proof: An *unlock* operation by the tread τ can either unlock the mutex m or activate one thread waiting in the m.waiting queue. Note that in the application, τ must own the mutex m when calling *unlock*. There are two main cases:

(I) The *unlock* unlocks the mutex m. In this case there are two sub-cases depending on the existence of concurrent *lock* operations (indicated by the value of m.count):
(i) No concurrent *lock*, i.e. the mutex state is $\langle 1, null \rangle$ before line U1. The CAS at line U1 changes the mutex state to $\langle 0, null \rangle$ and is therefore the LP.
(ii) There is at least one concurrent *lock*, i.e. the mutex state is $\langle > 1, null \rangle$ before line U1. In this case the *unlock* enters the *do_hand-off* procedure where H5 is the LP. Note that if the dequeue at line H2 or the CAS on line H8 succeeds, this *unlock* will activate a thread instead of unlocking the mutex. These cases are examined below.

(II) The *unlock* activates the waiting thread τ'. In this case the mutex state before line U1 is $\langle > 1, null \rangle$ indicating that there are concurrent *lock* operations. The *unlock* operation will enter the *do_hand-off* procedure to find the thread τ' to activate. There are three cases depending on the progress of the concurrent *lock* operation:
(i) The thread τ' has been enqueued on the m.waiting queue (line L4 in lock) before *unlock* reaches line H2. In this case the LP for *unlock* is the LP of the *dequeue* on line H2.
(ii) The thread τ' is enqueued on the m.waiting queue (line L4 in lock) after line H2 but before line H6. In this case the *is_empty* test on line H6 returns $False$ and the *unlock* will attempt to retake the hand-off token at line H8. This will succeed if and only if no other operation (e.g. a concurrent *lock* or *trylock*) took the token in the meanwhile. If the CAS at line H8 fails the *unlock* unlocked the mutex as per sub-case (ii) of case (I) above. If the CAS at line H8 succeeded the LP of *unlock* is the LP of the *dequeue* called on line H2 in the next iteration of the loop. Note that the loop can only be repeated once since to repeat m.waiting must be non-empty and *unlock* must retake responsibility of the mutex.
(iii) The thread τ' is enqueued on the m.waiting queue (line L4 in lock) after line H6. In this case *unlock* unlocked the mutex at line H5 as per sub-case (ii) of case (I) above.

□

4.3 Mutual exclusion properties

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

Theorem 5 (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*.*

Proof: A thread τ can become the owner of a mutex m by a *lock* operation or a successful *trylock* operation.

By Theorem 2, Theorem 3 and Theorem 4, *lock*, *trylock* and *unlock* are linearizable. Therefore, in a set of concurrent *lock* and *trylock* operations on an unlocked mutex, exactly one of them will take effect first, i.e. by changing the mutex state to locked, and the thread executing it becomes the owner of the mutex. When this thread unlocks the mutex by an *unlock* operation, then by Theorem 4 the ownership is either transfered to the first thread in the m.waiting queue, which is also activated, or, if there are no such threads, the mutex state changes to unlocked. □

Lemma 3 *No thread is left blocked in the waiting queue of an unlocked mutex* m *when all concurrent operations concerning* m *have completed.*

Proof: Recall from Definition 2 that a mutex m is locked if and only if m.count $>$ 0 and m.hand-off $= null$, so the mutex is unlocked if m.count $= 0$ or m.hand-off $\neq null$.

If m.count $= 0$ then there clearly cannot be any waiting threads, since the first action of a thread trying to acquire a mutex using *lock*, is to increase m.count.

Assume, towards a contradiction, that there are no uncompleted operations and there is a thread τ left in the m.waiting queue, m.count $\neq 0$ and m.hand-off $\neq null$. Consider the *lock* operation by the thread τ. It cannot have been the last operation to complete, because if m.hand-off was $\neq null$, the CAS at L8 would have succeeded and *lock* would dequeue the thread τ available in the m.waiting queue and activate it.

But if m.hand-off was $null$ when τ's *lock* operation completed, then there must have been some other uncompleted operation active inside a hand-off loop after that point, since that is the only place m.hand-off is set to something other than null (at line H5). However, for that operation to leave its hand-off loop and complete, it must find the m.waiting queue empty after setting m.hand-off (line H6). This contradicts our assumption that τ's *lock* operation completed before m.hand-off was set.

Thus, it is impossible that all operations on m completed leaving m unlocked and the thread τ in the m.waiting queue. □

Lemma 4 *A mutex is locked if and only if it is* owned *by a thread.*

Proof: The lemma follows from the Theorems 2, 3 and 4, i.e. linearizability of *lock*, *trylock* and *unlock*. □

Lemma 5 *A thread τ waiting to acquire a mutex m in a call to lock will at most have to wait for the thread currently owning m and all threads that have called lock on m before τ's call to lock enqueued τ on the m.waiting queue.*

Proof: Once the thread τ has been enqueued on the m.waiting queue (line L4) it only needs to wait for the threads ahead of it in the queue in addition to any current owner of the mutex before it will acquire the mutex. This is ensured by the *unlock* protocol that will activate the first thread in the m.waiting queue (lines H2 - H4). A *trylock* operation cannot bypass the waiting threads since m.count is nonzero and *unlock* only sets the m.hand-off if it finds the waiting queue to be empty. □

Theorem 6 (Liveness I) *A thread τ waiting to acquire a mutex m will eventually acquire the mutex once its lock operation has enqueued τ on the m.waiting queue.*

Proof: The theorem follows from Lemma 1, Lemma 3, Lemma 4 and Lemma 5. □

Theorem 7 (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.*

Proof: The theorem follows from the lock-free nature of the m.waiting queue and Theorem 6.

We know from Theorem 6 that once the thread has enqueued itself on the m.waiting queue it will not starve, so to starve it must not succeed to enter the m.waiting queue, that is, its enqueue operation must never complete.

Each time two or more operations on the lock-free queue interfere with each other, at least one of them make progress, so for one processor to never complete its operation, it will have to be interfered with by a concurrent successful operation every time it tries to progress. □

Theorem 8 (Fairness) *A thread τ wanting to acquire a mutex m will only have to wait for the threads whose lock operation enqueued them on the m.waiting queue before τ was enqueued there.*

Proof: The theorem follows from Lemma 5. □

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 of the impact of the lock-free mutex implementation. 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"[1] processes as *virtual processors* for running the threads. The implementation uses the lock-free queue in [37] 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 object implementation uses a test and test-and-set type spin-lock 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 well.

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 thread library 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.

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

Figure 5 Mutex performance in LFTHREADS and pthreads at high contention.

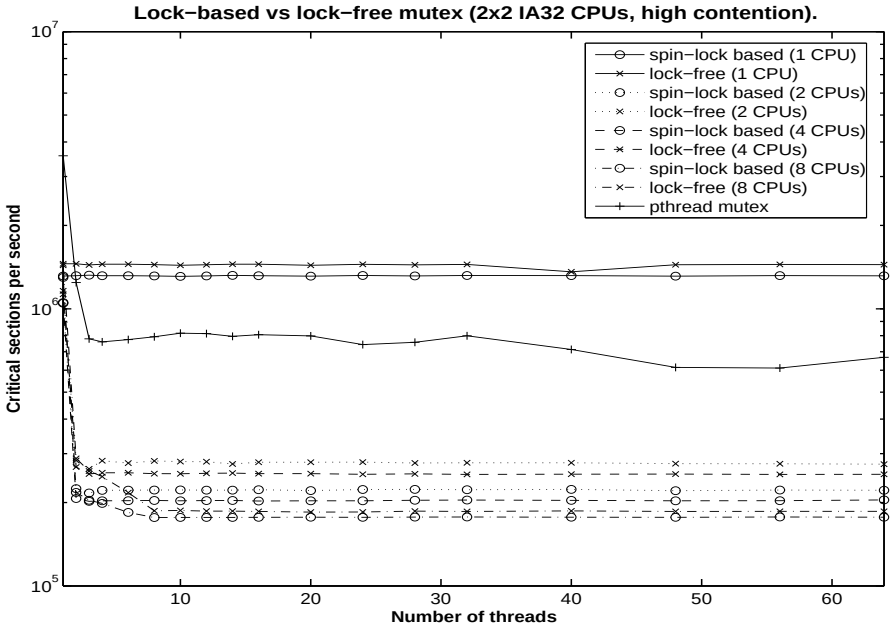
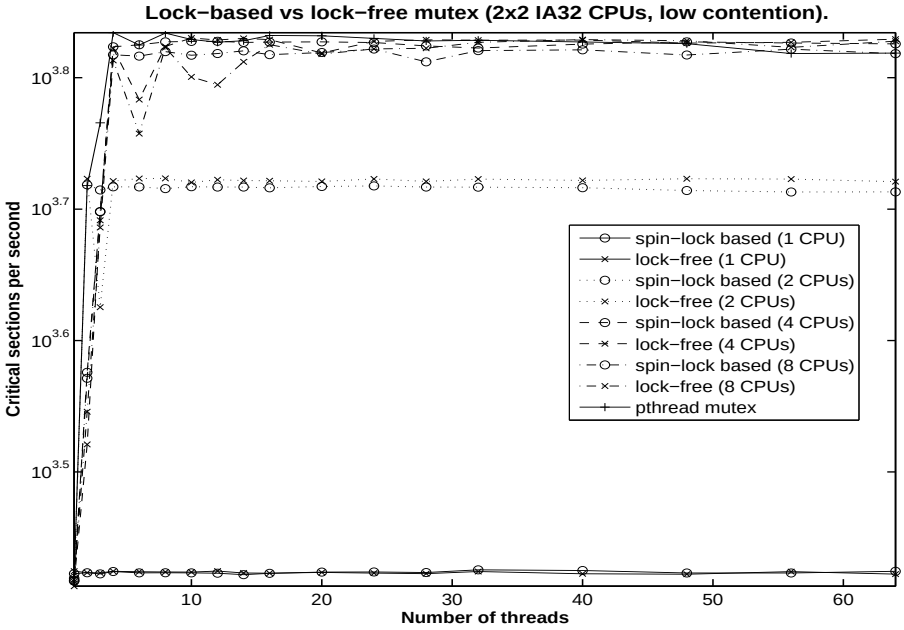


Figure 6 Mutex performance in LFTHREADS and pthreads at low contention.



- The spin-lock based mutex, using 1, 2, 4 and 8 virtual processors to run the threads.
- 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. This difference in scheduling makes it difficult to interpret the pthreads results with respect to the others; i.e. the pthreads results should be considered to be primarily for reference.

Each test configuration was run 10 times. The diagrams present the mean of these 10 runs.

High contention

In Figure 5 we show the microbenchmark results when all work is done inside the critical section, that is, the contention on the mutex is high. In this case the desired result would be that the throughput, i.e. the number of critical sections executed per second, for an implementation stays the same regardless of the number of threads or virtual (processors). This should 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. Further, going from a single processor to more than one processor for our thread library implies a cost since with more than one processor the thread contexts will have to be stored and restored much more often due to threads being blocked on the mutex. (Note that threads currently use non-preemptive scheduling in our implementation so with only one virtual processor the threads will run to completion one after the other without any extra blocking.) The results indicate that the lock-free mutex has less overhead than the lock-based one in similar configurations.

Low contention

In Figure 6 we show the results from a microbenchmark where 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 implementa-

tion, 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. Further, the lock-free mutex shows similar or higher throughput than the spin-lock-based one for the same number of virtual processors; it also shows comparable and even better performance than the pthread-based case when the number of threads is large and there are "enough" virtual processors (i.e. more than the physical processors).

Summarizing, we observe that the LFTHREADS thread library'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, besides offering the qualitative advantages in tolerance against slow, delayed or crashed threads, as discussed earlier in the paper.

6 Conclusions

In this paper we have presented the LFTHREADS library and the first lock-free implementation of a blocking synchronization primitive; as part of this contribution we have introduced the responsibility hand-off (RHO) synchronization method. Besides supporting a thread-library interface with fault-tolerance properties, we regard the RHO method as a conceptual contribution, which can be useful in multiprocessors and multicore systems in general.

We have implemented the LFTHREADS library on a PC multiprocessor platform with two Intel Xeon processors running the GNU/Linux operating system and using processes as virtual processors. The implementation does not need any modifications to the operating system kernel. Although our present implementation is done entirely at the user-level, the LFTHREADS 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.

This implementation constitutes a proof-of-concept of the lock-free implementation of the blocking primitive 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 performance figures. Moreover, this 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 applications from the SPLASH-2 suite.

Bibliography

[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 Transactions on Computer Systems*, pages 53–79, 1992.

[3] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270. SIGACT and SIGARCH, 1993. Extended abstract.

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

[5] B. Chen. Multiprocessing with the exokernel operating system. Master's thesis, Massachusetts Institute of Technology, 2000.

[6] D. L. Detlefs, P. A. Martin, M. Moir, and J. Guy L. Steele. Lock-free reference counting. In *Proceedings of the 20th annual ACM Symposium on Principles of distributed computing*, pages 190–199. ACM Press, 2001.

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

[8] 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.

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

[10] K. A. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

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

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

[13] 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.

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

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

[16] M. Herlihy. Software transactional memory package for C#, 2006. http://www.cs.brown.edu/people/mph/home.html.

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

[18] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, page 522. IEEE Computer Society, 2003.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101. ACM Press, 2003.

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

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

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

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

[24] L. Lamport. On interprocess communication–part i: Basic formalism, part ii: Algorithms. *Distributed Computing*, 1:77–101, 1986.

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

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

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

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

[29] 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.

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

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

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

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

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

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

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

[37] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143. ACM Press, 2001.

[38] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 212–222. ACM Press, 1992.

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

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

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

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available via WWW using the URL `http://www.mpi-inf.mpg.de`. If you have any questions concerning WWW access, please contact `reports@mpi-inf.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-inf.mpg.de`

.

MPI-I-2007-RG1-002	T. Hillenbrand, C. Weidenbach	Superposition for Finite Domains
MPI-I-2007-5-002	K. Berberich, S. Bedathur, T. Neumann, G. Weikum	A Time Machine for Text Search
MPI-I-2007-5-001	G. Kasneci, F.M. Suchanek, G. Ifrim, M. Ramanath, G. Weikum	NAGA: Searching and Ranking Knowledge
MPI-I-2007-4-006	C. Dyken, G. Ziegler, C. Theobalt, H. Seidel	GPU Marching Cubes on Shader Model 3.0 and 4.0
MPI-I-2007-4-005	T. Schultz, J. Weickert, H. Seidel	A Higher-Order Structure Tensor
MPI-I-2007-4-004	C. Stoll	A Volumetric Approach to Interactive Shape Editing
MPI-I-2007-4-003	R. Bargmann, V. Blanz, H. Seidel	A Nonlinear Viseme Model for Triphone-Based Speech Synthesis
MPI-I-2007-4-002	T. Langer, H. Seidel	Construction of Smooth Maps with Mean Value Coordinates
MPI-I-2007-4-001	J. Gall, B. Rosenhahn, H. Seidel	Clustered Stochastic Optimization for Object Recognition and Pose Estimation
MPI-I-2007-2-001	A. Podelski, S. Wagner	A Method and a Tool for Automatic Veriication of Region Stability for Hybrid Systems
MPI-I-2007-1-002	E. Althaus, S. Canzar	A Lagrangian relaxation approach for the multiple sequence alignment problem
MPI-I-2007-1-001	E. Berberich, L. Kettner	Linear-Time Reordering in a Sweep-line Algorithm for Algebraic Curves Intersecting in a Common Point
MPI-I-2006-5-006	G. Kasnec, F.M. Suchanek, G. Weikum	Yago - A Core of Semantic Knowledge
MPI-I-2006-5-005	R. Angelova, S. Siersdorfer	A Neighborhood-Based Approach for Clustering of Linked Document Collections
MPI-I-2006-5-004	F. Suchanek, G. Ifrim, G. Weikum	Combining Linguistic and Statistical Analysis to Extract Relations from Web Documents
MPI-I-2006-5-003	V. Scholz, M. Magnor	Garment Texture Editing in Monocular Video Sequences based on Color-Coded Printing Patterns
MPI-I-2006-5-002	H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum	IO-Top-k: Index-access Optimized Top-k Query Processing
MPI-I-2006-5-001	M. Bender, S. Michel, G. Weikum, P. Triantafilou	Overlap-Aware Global df Estimation in Distributed Information Retrieval Systems
MPI-I-2006-4-010	A. Belyaev, T. Langer, H. Seidel	Mean Value Coordinates for Arbitrary Spherical Polygons and Polyhedra in $\mathbb{R}^3$
MPI-I-2006-4-009	J. Gall, J. Potthoff, B. Rosenhahn, C. Schnoerr, H. Seidel	Interacting and Annealing Particle Filters: Mathematics and a Recipe for Applications
MPI-I-2006-4-008	I. Albrecht, M. Kipp, M. Neff, H. Seidel	Gesture Modeling and Animation by Imitation
MPI-I-2006-4-007	O. Schall, A. Belyaev, H. Seidel	Feature-preserving Non-local Denoising of Static and Time-varying Range Data
MPI-I-2006-4-006	C. Theobalt, N. Ahmed, H. Lensch, M. Magnor, H. Seidel	Enhanced Dynamic Reflectometry for Relightable Free-Viewpoint Video
MPI-I-2006-4-005	A. Belyaev, H. Seidel, S. Yoshizawa	Skeleton-driven Laplacian Mesh Deformations
MPI-I-2006-4-004	V. Havran, R. Herzog, H. Seidel	On Fast Construction of Spatial Hierarchies for Ray Tracing

MPI-I-2006-4-003	E. de Aguiar, R. Zayer, C. Theobalt, M. Magnor, H. Seidel	A Framework for Natural Animation of Digitized Models
MPI-I-2006-4-002	G. Ziegler, A. Tevs, C. Theobalt, H. Seidel	GPU Point List Generation through Histogram Pyramids
MPI-I-2006-4-001	A. Efremov, R. Mantiuk, K. Myszkowski, H. Seidel	Design and Evaluation of Backward Compatible High Dynamic Range Video Compression
MPI-I-2006-2-001	T. Wies, V. Kuncak, K. Zee, A. Podelski, M. Rinard	On Verifying Complex Properties using Symbolic Shape Analysis
MPI-I-2006-1-007	H. Bast, I. Weber, C.W. Mortensen	Output-Sensitive Autocompletion Search
MPI-I-2006-1-006	M. Kerber	Division-Free Computation of Subresultants Using Bezout Matrices
MPI-I-2006-1-005	A. Eigenwillig, L. Kettner, N. Wolpert	Snap Rounding of Bézier Curves
MPI-I-2006-1-004	S. Funke, S. Laue, R. Naujoks, L. Zvi	Power Assignment Problems in Wireless Communication
MPI-I-2005-5-002	S. Siersdorfer, G. Weikum	Automated Retraining Methods for Document Classification and their Parameter Tuning
MPI-I-2005-4-006	C. Fuchs, M. Goesele, T. Chen, H. Seidel	An Emperical Model for Heterogeneous Translucent Objects
MPI-I-2005-4-005	G. Krawczyk, M. Goesele, H. Seidel	Photometric Calibration of High Dynamic Range Cameras
MPI-I-2005-4-004	C. Theobalt, N. Ahmed, E. De Aguiar, G. Ziegler, H. Lensch, M.A. Magnor, H. Seidel	Joint Motion and Reflectance Capture for Creating Relightable 3D Videos
MPI-I-2005-4-003	T. Langer, A.G. Belyaev, H. Seidel	Analysis and Design of Discrete Normals and Curvatures
MPI-I-2005-4-002	O. Schall, A. Belyaev, H. Seidel	Sparse Meshing of Uncertain and Noisy Surface Scattered Data
MPI-I-2005-4-001	M. Fuchs, V. Blanz, H. Lensch, H. Seidel	Reflectance from Images: A Model-Based Approach for Human Faces
MPI-I-2005-2-004	Y. Kazakov	A Framework of Refutational Theorem Proving for Saturation-Based Decision Procedures
MPI-I-2005-2-003	H.d. Nivelle	Using Resolution as a Decision Procedure
MPI-I-2005-2-002	P. Maier, W. Charatonik, L. Georgieva	Bounded Model Checking of Pointer Programs
MPI-I-2005-2-001	J. Hoffmann, C. Gomes, B. Selman	Bottleneck Behavior in CNF Formulas
MPI-I-2005-1-008	C. Gotsman, K. Kaligosi, K. Mehlhorn, D. Michail, E. Pyrga	Cycle Bases of Graphs and Sampled Manifolds
MPI-I-2005-1-007	I. Katriel, M. Kutz	A Faster Algorithm for Computing a Longest Common Increasing Subsequence
MPI-I-2005-1-003	S. Baswana, K. Telikepalli	Improved Algorithms for All-Pairs Approximate Shortest Paths in Weighted Graphs
MPI-I-2005-1-002	I. Katriel, M. Kutz, M. Skutella	Reachability Substitutes for Planar Digraphs
MPI-I-2005-1-001	D. Michail	Rank-Maximal through Maximum Weight Matchings
MPI-I-2004-NWG3-001	M. Magnor	Axisymmetric Reconstruction and 3D Visualization of Bipolar Planetary Nebulae
MPI-I-2004-NWG1-001	B. Blanchet	Automatic Proof of Strong Secrecy for Security Protocols
MPI-I-2004-5-001	S. Siersdorfer, S. Sizov, G. Weikum	Goal-oriented Methods and Meta Methods for Document Classification and their Parameter Tuning
MPI-I-2004-4-006	K. Dmitriev, V. Havran, H. Seidel	Faster Ray Tracing with SIMD Shaft Culling
MPI-I-2004-4-005	I.P. Ivrissimtzis, W.-. Jeong, S. Lee, Y.a. Lee, H.-. Seidel	Neural Meshes: Surface Reconstruction with a Learning Algorithm