Tidex - A Mutual Exclusion Lock

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Abstract

Several basic mutual exclusion lock algorithms are known, with one of the simplest being the Ticket Lock. We present a new mutual exclusion lock with properties similar to the Ticket Lock but using `atomic_exchange()` instead of `atomic_fetch_add()` that can be more efficient on systems without a native instruction for `atomic_fetch_add()`, or in which the native instruction for `atomic_exchange()` is faster than the one for `atomic_fetch_add()`. Similarly to the Ticket Lock, our lock has small memory foot print, is extremely simple, respects FIFO order, and provides starvation freedom in architectures that implement `atomic_exchange()` as a single instruction, like x86.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Mutual Exclusion

General Terms Algorithms, Design, Performance

Keywords mutual exclusion, ticket lock, locks

1. Introduction

Spin Locks are a class of Locks that provide mutual exclusion and remain an important and widely used synchronization mechanism for thread-safe concurrent programming. One example of a spin lock is a "‘test and set"’ implementation that will simply loop, attempting to use an atomic instruction to change a memory word unlocked to locked state, either using an `atomic_compare_exchange()` [2] or `atomic_exchange()` [3]. In such an implementation, where all threads spin (loop, busy-waiting) on the same lock variable, we say this technique uses *global spinning*.

A Ticket Lock [6] is another example of a simple global spinning lock which in its basic form consists of two variables, a `ticket` and a `grant`. Arriving threads atomically fetch-and-add the `ticket` and then spin until the `grant` variable matches the value returned by the atomic fetch-and-add primitive. With this, the thread has *acquired the lock* and may safely enter the critical section. After exiting the critical section, the thread *releases the lock* by advacing the `grant` variable. This can be done with a simple store operation, or in the C11/C++1x memory model [1], a store with release (`atomic_store()`). By advancing the `grant`, the lock can now be acquired by the next thread, if any. This behavior provides FIFO ordering, unlike the test-and-set lock. A variant of the Ticket Lock is the Partioned Ticket Lock [5] which provides FIFO ordering but with semi-local spinning.

The Tidex Lock is composed of three variables: a `ticket`, a `grant` and a `nextGrant`, where the first two are sequentially consistent atomics, and the `nextGrant` is a *non-atomic variable*. Each thread needs two unique identifiers, which can easily be obtained from `pthread_self()` and its negative, but other techniques can be devised, for example using a one-time assigned `atomic_fetch_add()` [4] from a global variable, which is then stored in a thread-local variable and shared among all Tidex lock instances. For simplicity, we will henceforth refer to these two identifiers as `TID` and `-TID`.

A thread arriving to acquire the lock will start by checking with a relaxed load whether the `ticket` is currently set to its `TID`. If it is currently at `TID` then it must use `-TID`, otherwise, it is safe to use `TID` as the identifier to acquire the lock. The thread will then do `atomic_exchange()` [3] on the `ticket` variable, replacing the current value with its own identifier, and saving the previous identifier on a local variable. Then, it will spin until the `grant` variable matches the identifier returned by the `atomic_exchange()` operation. Once they match, the lock has been acquired by the current thread, and all it needs to do is to set the `nextGrant` to the identifier that was put in `ticket` (can be `TID` or `-TID`), so that the unlock procedure can set the `grant` to that value, and in fact it is all that the unlock procedure does. The algorithm for the Tidex lock is shown in Algorithm 1 in C11 code.
Notice that the thread attempting to acquire the lock does a single `atomic_exchange()`, unlike test-and-set locks which do multiple `atomic_exchange()` calls in a loop until the lock is acquired.

In line 9 of Algorithm 1, there is an optimization, where the `ticket` variable is read with a relaxed-load [1] instead of an acquire-load. This is a valid optimization because the thread is only interested in knowing if it was itself the last thread to acquire the lock, whether itself acquired this particular lock instance with the `TID` or `-TID` identifier, so it only has to distinguish between these cases, and the contents of the `ticket` variable will in that case be *up to date* because it is the same thread.
It is only in line 16 of Algorithm 1 that the `nextGrant` can be set to the value set by `atomic_exchange()` because it is only then that the lock has been acquired, thus providing a guarantee of mutual exclusion so that we are certain that the only thread writing into `nextGrant` is the current thread holding the lock.
The reason why the `nextGrant` variable is needed, is so that the unlock procedure can set the `grant` to the same identifier that was put in `ticket`, but this variable can be discarded if instead we return such a value from the `tidex_lock()` function and then pass it as an argument to `tidex_unlock()`, thus saving one store per lock/unlock.

In Algorithm 1 we show a possible implementation of the `trylock()` method for Tidex. We start by checking in line 26

Algorithm 1 Tidex Mutex Algorithm

```
1   typedef struct {
2       atomic_llong ticket;
3       atomic_llong grant;
4       long long nextGrant;
5   } tidex_t;
6
7   void tidex_lock(tidex_t * self) {
8       long long mytid = (long long)pthread_self();
9       if (atomic_load_explicit(&self->grant,
10          memory_order_relaxed) == mytid) mytid = -mytid;
11      long long prevtid = atomic_exchange(&self->ticket, mytid);
12      while (atomic_load(&self->grant) != prevtid) {
13          sched_yield();
14      }
15      // Lock has now been acquired by this thread
16      self->nextGrant = mytid;
17  }
18
19  void tidex_unlock(tidex_t * self) {
20     atomic_store(&self->grant, self->nextGrant);
21  }
22
23  int tidex_mutex_trylock(tidex_mutex_t * self) {
24      long long localG = atomic_load(&self->grant);
25      long long localT = atomic_load_explicit(&self->ticket, memory_order_relaxed);
26      if (localG != localT) return EBUSY;
27      long long mytid = (long long)pthread_self();
28      if (localG == mytid) mytid = -mytid;
29      if (!atomic_compare_exchange_strong(&self->ticket, &localT, mytid)) return EBUSY;
30      // Lock has been acquired
31      self->nextGrant = mytid;
32      return 0;
33  }
```

whether there is already another thread currently holding the lock, thus causing `ticket` and `grant` to be different. If the lock is available, an `atomic_compare_exchange()` operation is attempted, in order to modify `ticket` using the same logic as in `tidex_lock()`. The reason why we can't use an `atomic_exchange()` operation is due to the case when another thread acquires the lock and changes `ticket` between the check on line 26 and this thread's attempt to acquire the lock, which would then make it difficult or impossible to *abort/rollback* the intent to acquire the lock by this thread. This approach to `trylock()` is the same as the one on the Ticket Lock algorithm, which replaces an `atomic_fetch_add()` with an `atomic_compare_exchange()`.

2. Performance

Figure 1 shows the number of operations per second as a function of the number of threads on a microbenchmark that iterates for 30 seconds. Each iteration acquires the lock, reads all items in an array of 256 integers (critical section), then releases the lock, and executes a non-critical section that takes about 10 times longer to complete than the critical section. Only one array and one lock instance are used, and there are no writes in the critical section. The benchmark was ran on a dual Opteron 6272 with a total of 32 cores, running Windows 7 Professional SP1, using GCC 4.9.2 for Windows (MinGW). We compare the performance of a simple Ticket Lock and the Tidex Lock.

In terms of throughput, on x86 there is no advantage, due to both `atomic_exchange()` and `atomic_fetch_add()` being implemented as a single instruction, namely XCHG and XADD, respectively, and due to their relative time of execution. Other CPU architectures may have different performance characteristics.

Unlike the Ticket Lock, there is no need to concern with overflow of the `ticket` or `grant`, because these words need to be big enough to hold a unique thread identifier, i.e. the number of bits must be enough to represent twice the number of threads.

Source code in C11, and Java is available as part of the ConcurrencyFreaks library [7].

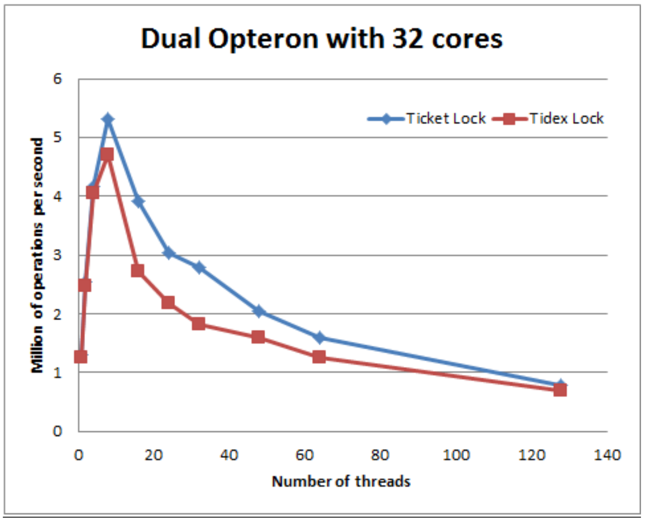

Figure 1. Lock scalability showing the total number of operations as we increase the number of threads contending on the lock.

References

[1] CPP-ISO-committee. C++ Memory Order. `http://en.cppreference.com/w/c/atomic/memory\_order`, 2013.

[2] CPP-ISO-committee. atomic_compare_exchange. `http://en.cppreference.com/w/c/atomic/atomic\_compare\_exchange`, 2014.

[3] CPP-ISO-committee. atomic_exchange. `http://en.cppreference.com/w/c/atomic/atomic\_exchange`, 2014.

[4] CPP-ISO-committee. atomic_fetch_add. `http://en.cppreference.com/w/c/atomic/atomic\_fetch\_add`, 2014.

[5] D. Dice. Brief announcement: a partitioned ticket lock. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 309–310. ACM, 2011.

[6] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[7] P. Ramalhete and A. Correia. Concurrency Freaks. `https://github.com/pramalhe/ConcurrencyFreaks`, 2015.