October 2021

Scalable, Highly Performant, Reader/Writer Lock Using Restartable Sequences

N/A

Scalable, Highly Performant, Reader/Writer Lock Using Restartable Sequences

ABSTRACT

Traditional shared locks for synchronizing mostly read-only data are expensive and scale poorly with the number of threads or processor cores. This disclosure describes a scalable, low-cost, low-contention, shared lock (a mutex implementation) that behaves well under cases of reasonable contention and under temporarily write-dominated scenarios. The described shared lock, also known as counting mutex, is based on restartable sequences and fast fences.

KEYWORDS

- Shared lock
- Reader lock
- Writer lock
- Mutex
- Restartable sequence
- Fast fences
- Cache miss
- Cache stall

BACKGROUND

Traditional shared locks for synchronizing mostly read-only data are expensive. Implementations such as `absl::Mutex::ReaderLock` require atomic operations on a shared mutex state [1]. These operations also scale poorly since each thread performing (locked) atomic operations on the same shared state will incur cache misses and stall as multiple threads compete on the shared state. With the number of cores per socket increasing with each new generation of processors and sharded lower level caches becoming more common, the compute cost and contention for shared locks increases.

The main cost factors of (reader) locks are cycles spent in (locked) compare and swap operations, flush of local write buffers, and cache-to-cache transfer of the mutex state across cores on concurrent locks. Another detrimental effect is false sharing. An application serving

infrequently updated data protected by a read lock incurs the cost of the read lock and cache misses from data sharing the mutex cache line.

Read-copy-update (RCU), which is a Linux technique to synchronize locks, has certain shortcomings that make it unviable for many use cases. These include:

- RCU often requires a substantial redesign and refactoring of code.
- RCU requires copy-on-write (COW) logic, which may be infeasible if the synchronized corpus is large. For example, if the synchronized data is a large hash map with entries infrequently being added, updated, or removed, the overhead and extra memory required (peak RAM times two) of the COW logic is substantial.
- RCU requires application discipline to avoid long-held snapshots; the absence of such discipline can lead to many copies of the RCU-managed data to remain unreleased.

Other solutions shard traditional mutexes or states on a per-thread or per-CPU basis. While these reduce the contention of multiple threads competing on the same cache lines or data, they remain expensive operations. For example, even a non-contended, single-threaded atomic compare and swap on a modern processor is an order of magnitude (10x cycles or more) slower than regular load/store operations.

DESCRIPTION

This disclosure describes a scalable, low-cost, low-contention, shared lock (a mutex implementation) that behaves well under cases of reasonable contention and under temporarily write-dominated scenarios. The described shared lock, also known as counting mutex, is based on restartable sequences and fast fences. Some of its features include:

- Fast path execution on platforms that support restartable sequences (RSEQ) minimizes the actual RSEQ operations to two loads, one store, and one predicted branch.

- Fast path execution is synchronized using RSEQ fences, repointing the per-CPU base pointer to divert the single slow path branch of the RSEQ execution.
- Each per-CPU state is a combined counter/state, which enables writers to identify CPUs that may hold or have held per-CPU locks, and thereby reduces the impact (blast rate) of the CPU fences.
- Writer locks enter a futex wait state in the presence of readers, where unlocking readers are diverted to a global counter, signaled by the last reader reducing the wait counter to 0.
- An efficient fallback implementation when not using RSEQ that scales well and is on-par or better than standard (`absl::Mutex`) performance in a single-threaded use case.
- The reader lock is scalable in the sense that it minimizes the number of bus locks in the fast path; incurs or causes a minimum number of cache misses; scales well with the number of cores or threads; and approximates an O(1) cost per read lock regardless of concurrency.

For the use case, assume that read locks do not contend with write locks, e.g., write locks are rare. Write locks can also be relatively expensive as compared to standard `absl::Mutex` write locks.

Conceptually, the described shared-lock techniques are based on a RSEQ `percpu` *lock counter*. Each thread acquiring a read lock increases the `percpu` counter and, upon releasing the lock, decreases the `percpu` counter. Updates to the counter take place inside an *RSEQ critical section* to enable reader locks to use cheap, non-atomic loads and stores.

The least significant bit (LSB) of each counter indicates if that counter is active. Active CPU slots hold odd values and inactive CPU slots hold even values. Counters have a default

value of zero. Reader locks are increased and decreased by a value of two: counter increments and decrements do not affect the active state of that counter.

Counters can be atomic `uint32_t` values, under the reasonable assumption that there are at any time fewer than `uint32_max` threads concurrently holding a reader lock. Unsigned values signify that a thread holding a reader lock can be rescheduled to a different CPU before the reader lock is released, implying that slots can overflow or underflow. Also, using unsigned values provides defined behavior for underflow and overflow, and the cumulative sum of all counters (bar the LSB) remains correct.

Counters are organized inside *slabs* and indexed using a constant multiplication factor, such that counters are cache aligned per CPU, guaranteeing that each `percpu` counter neither shares a cache line with another `percpu` counter nor incurs cache misses.

Preventing new reader locks from occurring is done in the following way. The class has an atomic pointer to the `percpu` counter slab, which readers load inside the critical section. Write locks (holding the `absl::Mutex`) change this pointer to a singleton *always inactive* slab. Any reader not currently inside an RSEQ critical section is thus forced onto the slow path as it tries to increment that singleton counter on each subsequent lock or unlock attempt.

The same mutex must be held to transition a `percpu` counter from *inactive* to *active*. This guarantees that no new read locks can be obtained once the mutex is obtained by a thread obtaining a write lock. New reader locks then block and wait on the mutex held by the concurrent write lock. For reader *unlocks* disallowed from blocking, a regular single `pending_` atomic counter is used to count reader unlocks on inactive slots. Writers detecting concurrent reader locks will use a futex wait on this `pending_` counter: *in-flight* read locks are guaranteed to decrease this value.

Obtaining a read lock

As explained earlier, reader locks don't contend, reducing the fast path inside the critical section to two loads, a branch, an addition, and single store.

```
void ReaderLock()
  RSEQ {
    counter = counter_.load(std::memory_order_acquire);
    value = counter[CUR_CPU_TO_SLOT];
    if ((value & 1) == 0) goto slow_path;
    counter_[CUR_CPU_TO_SLOT] = value + 2;
    return;
  }
slow_path:
  MutexLock lock(&mu_);
  RSEQ {
    value = counter_[CUR_CPU_TO_SLOT];
    counter_[CUR_CPU] = (value | 1)  + 2;
  }
  reader_locks_.store(true, std::memory_order_release);
}

void ReaderUnlock()
  RSEQ {
    value = counter_[CUR_CPU_TO_SLOT];
    if ((value & 1) == 0) goto slow_path;
    counter_[CUR_CPU] = value - 2;
    return;
  }
slow_path:
  result = pending_.fetch_sub(2);
  if (result == 2) Futex::Signal(&pending_);

private:
  absl::Mutex mu_;
  Handle_ = AllocHandle();
  std::atomic<uint32_t>* counter_{handle_.ptr};
  std::atomic<uint32_t> pending_{0};
  std::atomic<bool> reader_locks_{false};
  static uint32_t* null_counter_;
```

Fig. 1: Obtaining a read lock

Fig. 1 illustrates an example of code to obtain a read lock. The `reader_locks_` state defends against the lock being used in (temporary) write lock dominated use cases: the `WriteLock` implementation (described further below) acquires a lock on the mutex and only exercises the slow writer lock path if it detects the presence of any reader lock. This makes `WriteLock` perform identical to a regular `absl::Mutex` in the absence of reader locks.

Obtaining a write lock

```
WriterLock() {
  mu_.WriterLock();
  if (reader_locks_.load(std::memory_order_acquire)) {
    counter_.store(null_counter_);
    for (int cpu = 0; cpu < NumCPUs(); ++cpu) {
      if ((counter_[CUR_CPU] & 1) != 0) {
        FenceCpu(cpu);

        uint32_t value = counter_[CUR_CPU];
        assert(value & 1);
        counter_[CUR_CPU] = 0;
        pending_ += value - 1;
      }
    }
    pending += pending_.fetch_add(pending);
    while (pending != 0) {
       FutexWait(&pending_, pending);
       pending = pending_.load();
    }
  }
}

WriterUnlock()
  if (reader_locks_.load(std::memory_order_acquire)) {
    counter_ = handle_.ptr;
    reader_locks_.store(false, std::memory_order_release);
  }
  mu_.WriterUnlock();
```

Fig. 2: Obtaining a write lock

Fig. 2 illustrates an example of code to obtain a write lock.

The described techniques provide substantial improvement in performance for single thread (uncontended) as well as multi-threaded concurrent reader-lock / reader-unlock calls, and for single thread (non contended) as well as contended WriterLock / WriterUnlock calls

CONCLUSION

This disclosure describes a scalable, low-cost, low-contention, shared lock (a mutex implementation) that behaves well under cases of reasonable contention and under temporarily write-dominated scenarios. The described shared lock, also known as counting mutex, is based on restartable sequences and fast fences.

REFERENCES

[1] “abseil / Synchronization Library”, available online at https://abseil.io/docs/cpp/guides/synchronization#abslmutex-and-stdmutex accessed Sep. 23, 2021.