TWA – Ticket Locks Augmented with a Waiting Array

Dave Dice  
Oracle Labs  
dave.dice@oracle.com

Alex Kogan  
Oracle Labs  
alex.kogan@oracle.com

Abstract

The classic ticket lock[13, 18, 20] consists of ticket and grant fields. Arriving threads atomically fetch-and-increment ticket and then wait for grant to become equal to the value returned by the fetch-and-increment primitive, at which point the thread holds the lock. The corresponding unlock operation simply increments grant. This simple design has short code paths and fast handover (transfer of ownership) under light contention, but may suffer degraded scalability under high contention when multiple threads busy wait on the grant field – so-called global spinning. The MCS lock[18] is the usual alternative to ticket locks, performing better under high contention, but also having a more complex path and often lagging behind ticket locks under no or light contention. In MCS, arriving threads use an atomic operation to append an element to a queue of waiting threads, and then busy wait on a field in that element, using local spinning. The unlock operator must identify the successor, if any, and then store to the location where the successor busy waits, passing ownership. The handover path is longer than that of ticket locks and accesses more distinct shared locations. Developers face a dilemma in choosing between MCS locks and ticket locks.

We propose a variation on ticket locks where long-term waiting threads wait on locations in a waiting array instead of busy waiting on the grant field. The single waiting array is shared among all locks. Short-term waiting is accomplished in the usual manner on the grant field. The resulting algorithm, TWA, improves on ticket locks by limiting the number of threads spinning on the grant field at any given time, reducing the number of remote caches requiring invalidation from the store that releases the lock. In turn, this accelerates handover, and since the lock is held throughout the handover operation, scalability improves. Under light or no contention, TWA yields performance comparable to the classic ticket lock, avoiding the complexity and extra accesses incurred by MCS locks in the handover path, but providing performance above or beyond that of MCS at high contention.

CSC Concepts  • Software and its engineering → Multithreading; Mutual exclusion; Concurrency control; Process synchronization;

Keywords  • Locks, Mutexes, Mutual Exclusion, Synchronization, Concurrency Control

1 Introduction

1.1 Ticket locks vs MCS

The classic ticket lock is compact and has a very simple design. The acquisition path requires only one atomic operation – a fetch-and-add to increment the ticket – and the unlock path requires no atomics. On Intel systems, fetch-and-add is implemented via the LOCK: XADD instruction so the doorway phase is wait-free. Under light or no contention, the handover latency, defined as the time between the call to unlock and the time a successor is enabled to enter the critical section, is low. Handover time impacts the scalability as the lock is held throughout handover, increasing the effective length of the critical section [3, 12]. Typical implementations use 32-bit integers for the ticket and grant variable. Rollover not a concern as long as the number of concurrently waiting threads on a given lock never exceeds $2^{32} - 1$. A ticket lock is in unlocked state when ticket and grant are equal. Otherwise the lock is held, and the number of waiters is given by Ticket − Grant − 1. Ignoring numeric rollover, grant always lags or is equal to ticket. The increment operation in unlock either passes ownership to the immediate successor, if any, and otherwise sets the state to unlocked.

Ticket locks suffer, however, from a key scalability impediment. All threads waiting for a particular lock will busy wait on that lock’s grant field. An unlock operation, when it increments grant, invalidates the cache line underlying grant for all remote caches where waiting threads are scheduled. In turn, this negatively impacts scalability by retarding the handover step. We say that ticket locks use global spinning, as all waiting threads monitor the central lock-specific grant variable.

In Figure-1 we show the impact of passive readers on a single active writer. We refer to the number of participating caches as the invalidation diameter[10]. The Invalidation Diameter benchmark spawns T concurrent threads, with T shown on the X-axis. A single writer threads loops, using an atomic fetch-and-add primitive to update a shared location. The other T − 1 threads are readers. They loop, fetching the value of that location. The shared variable is sequestered to avoid false sharing and is the sole occupant of its underlying cache sector. We present the throughput rate of the writer on the Y-axis. As can be seen, as we increase the number of concurrent readers, the writer’s progress is slowed. This

---

1 Developers assume this constraint is always satisfied – having more that $2^{32} - 1$ waiting threads is not considered a practical concern.
scenario models the situation in ticket locks where multiple waiting threads monitor the grant which is updated by the current owner during handover. The benchmark reports the writer’s throughput at the end of a 10 second measurement interval. The data exhibited high variance due to the NUMA placement vagaries of the threads and the home node of the variable. As such, for each data point show, we took the median of 100 individual runs, reflecting a realistic set of samples. The system-under-test is described in detail in §3.

![Graph showing Invalidation Diameter](image)

**Figure 1. Invalidation Diameter**

Under classic MCS, arriving threads append an element to the tail of the list of waiting threads and then busy wait on a flag within that element. The lock’s tail variable is explicit and the head – the current owner – is implicit. When the owner releases the lock it reclaims the element it originally enqueued and sets the flag in the next element, passing ownership. MCS uses so-called local waiting. At most one thread is waiting on a given location at any one time. As such, an unlock operation will normally need to invalidate just one location – the flag where the successor busy waits. Under contention, the unlock operator must fetch the address of the successor node from its own element, and then store into the flag in the successor’s element, accessing two distinct cache lines, and incurring a dependent load to reach the successor. (Ticket locks require no such indirection or dependent loads in the unlock path).

The standard POSIX pthread_mutex_lock and pthread_mutex_unlock operators do not require scoped or lexically balanced locking. MCS implementations exposed via the POSIX interface will typically allocate elements – so-called “queue nodes” – from thread-local free lists. One queue node instance is required for each lock a thread currently holds, and an additional queue node is required while a thread is waiting on a lock. (The MCS “K42” variant [17, 21] allows queue nodes to be allocated on stack – they are required only while a thread waits – but at the cost of a longer path with more accesses to shared locations).

The POSIX interface does not provide any means to pass information from a lock operation to the corresponding unlock operator. As such, the address of the MCS queue node associated with the owner thread is usually recorded in the lock instance so it can be used in the subsequent unlock operation to identify the successor.

Ticket locks (and TWA) avoid the need for queue nodes and the management thereof. The queue of waiting threads is implicit in ticket locks, and explicit in MCS. MCS, ticket locks and TWA all provide strict FIFO admission order.

Ticket locks are usually a better choice under light or contention, while MCS locks are more suitable under heavy contention³ [4, 5].

### 1.2 The TWA Algorithm

TWA builds directly on ticket locks. We add a new waiting array for long-term waiting. The array is shared amongst all threads and TWA locks in an address space. Arriving threads use an atomic fetch-and-increment to advance the ticket value, yielding the lock request’s assigned ticket value, and then fetch grant. If the difference is 0 then we have have an uncontended acquisition and the thread may enter the critical section immediately. (This case is sometimes referred to as the lock acquisition fast-path). Otherwise TWA compares the difference to the LongTermThreshold parameter. If the difference exceeds LongTermThreshold then the thread enters the long-term waiting phase. Otherwise it proceeds to the short-term waiting phase, which is identical to that of normal ticket locks; the waiting thread simply waits for grant to become equal to the ticket value assigned to the thread. While LongTermThreshold is a tunable parameter in our implementation, we found a value of 1 to be suitable for all environments, ensuring that only the immediate successor waits in short-term mode. All data reported below uses a value of 1.

Threads in the long-term waiting phase first hash their assigned ticket value to form an index into the waiting array. Using this index, they fetch the value from the array and then recheck the value of grant. (The recheck step is needed to avoid races between lock and unlock operations). If the observed grant value changed, they recheck the difference between that new value and their assigned ticket value, and decide once again on short-term versus long-term waiting. If grant was unchanged, the thread then busy waits for the waiting array value to change, at which point it reevaluates

³The linux kernel switched from ticket locks to MCS-based locks in 2014 [6].
grant. When grant is found to be sufficiently near the assigned ticket value, the thread reverts to normal short-term waiting. The values found in the waiting array have no particular meaning, except to conservatively indicate that a grant value that maps to that index has changed, and rechecking of grant is required for waiters on that index. As rollover is a concern in the waiting array, we use 64-bit integers, so in practice, rollover never occurs.

The unlock operator increments grant as usual from \( U \) to \( U + 1 \) and then uses an atomic fetch-and-add to increment the location in the waiting array that corresponds to threads waiting on ticket value \( U + 1 + \text{LongTermThreshold} \), notifying long-term threads, if any, that they should recheck grant. We observe that this change increases the path length in the unlock operator, but crucially the store that effects handover, which is accomplished by a non-atomic increment of grant, happens first. Given a \( \text{LongTermThreshold} \) value of 1, we expect at most one thread, the immediate successor, to be waiting on grant. Updating the waiting array occurs after handover and outside the critical section, and does not influence scalability.

All our experiments use a waiting array with 1024 elements, although ideally, we believe the waiting array should be sized as a function of the number of CPUs in the system. (A similar approach is used to size the futex hash table array in the Linux kernel.) Hash collisions in the table are possible but benign, at worst causing unnecessary rechecking of the grant field. Collisions are a performance and quality-of-implementation concern that does not impact correctness. Large waiting array tables will reduce the collisions rate but might increase cache pressure. We note that the odds of inter-lock collision are equivalent to those given by the “Birthday Paradox” [22]. Our hash function is cache-aware and intentionally designed to map adjacent ticket values to different 128-byte cache sectors underlying the waiting array, to reduce false sharing among long-term waiters. We multiply the ticket value by 127 (a prime), EXCLUSIVE-OR that result with the address of the lock, and then mask with 1024 – 1 to form an index. Multiplication by 127 is easily strength-reduced to a shift and subtract. We include the lock address into our deterministic hash to avoid the situation where two locks might operate in an entrained fashion, with ticket and grant values moving in near unison, and thus suffer from excessive inter-lock collisions. A given lock address and ticket value pair always hashes to the same index. The hash computed in the unlock operator must target the same index as the corresponding hash in the long-term waiting path. We also note that near collisions can result in false sharing, when two accesses map to distinct words in the same cache sector.

TWA leaves the structure of the ticket lock unchanged, allowing for easy adoption. As the instance size remains the same, the only additional space cost for TWA is the waiting array, which is shared.

The TWA fast-path for acquisition remains unchanged relative to ticket locks. The unlock path adds precautionary increment of the waiting array, to notify any long-term waiters that they should transition from long-term to short-term. We note that TWA doesn’t reduce overall coherence traffic, but does act to reduce coherence traffic in the critical handover path, constraining the invalidation diameter of the store in unlock that accomplishes handover. TWA captures the desirable performance aspects of both MCS locks and ticket locks.

The fetch-and-increment of the waiting array element in the unlock operator can be avoided as follows. The waiting array contains unique thread identities instead of notification counters. Threads start a long-term waiting phase by storing their non-zero identity into the array and then recheck the value of grant. They then busy-wait while the array element remains equal to their own identity. When the element changes, the thread shifts to classic short-term waiting. The unlock operator simply stores 0 into the waiting array instead of incrementing the location. While this approach eliminates the atomic fetch-and-add in the unlock path, it also increases write traffic into the shared array, as threads entering the long-term write phase must store their unique identity.

Listing-1 depicts a pseudo-code implementation of the TWA algorithm. Lines 7 through 16 reflect the classic ticket lock algorithm and lines 20 through 71 show TWA. TWA extends the existing ticket lock algorithm by adding lines 41 through 57 for long-term waiting, and line 71 to notify long-term waiters to shift to classic short-term waiting.

1.3 Example Scenario – TWA in Action

1. Initially the lock is in unlocked state with Ticket and Grant both 0.
2. Thread \( T1 \) arrives at Listing-1 line 34 attempting to acquire the lock. \( T1 \) increments Ticket from 0 to 1, and the atomic FetchAdd operator returns the original value of 0 into the local variable \( tx \), which holds the assigned ticket value for the locking request. At line 36 \( T1 \) then fetches Grant observing a value of 0. Since \( tx \) equals that fetched value, we have uncontended lock acquisition. \( T1 \) now holds the lock and can enter the the critical section immediately, without waiting, via the fast path at line 39.
3. Thread \( T2 \) now arrives and tries to acquire the lock. The FetchAdd operator advances Ticket from 1 to 2 and returns 1, the assigned ticket, into \( tx \) at line 35. \( T2 \) fetches Grant and notes that \( tx \) differs from that value by 1.

\[ \text{dx} \] variable holds that computed difference, which reflects the number of threads between the requester and the head of the logical queue, which is the owner. \( T2 \)

https://blog.stgolabs.net/2014/01/futexes-and-hash-table-collisions.html
has encountered contention and must wait. The difference is only 1, and T2 will be the immediate successor, so T2 proceeds to line 60 for short-term waiting similar to that used in classic ticket locks shown at line 10. T2 waits for the Grant field to become 1.

Thread T3 arrives and advances Ticket from 2 to 3, with the FetchAdd operator returning 2 as the assigned ticket. The difference between that value (2) and the value of Grant(0) fetched at line 64 exceeds the LongTermThreshold (1), so T3 enters the path for long-term waiting at line 49. T3 hashes its observed ticket value of 2 into an index at, say 100, in the long-term waiting array and then fetches from WaitArray[100] observing U. To recover from potential races with threads in the unlock path, T3 rechecks that the Grant variable remains unchanged (0) at line 49 and that the thread should continue with long-term waiting. Thread T3 busy waits at lines 52-53 on the WaitArray value.

Thread T4 arrives, advances Ticket from 3 to 4, obtaining a value in its tx variable of 3. Similar to T3, T4 enters the long-term. T4 hashes its assigned ticket value of 3 yielding an index of, say, 207, and fetches WaitArray[207] observing V. T4 then busy waits, waiting for WaitArray[207] to change from V to any other value.

Thread T1 now releases the lock, calling TicketRelease at line 63. T1 increments Grant from 0 to 1 at line 67, passing ownership to T2 and sets local variable k to the new value (1).

Thread T2 waiting at lines 60-61 notices that Grant changed to match its tx value. T2 is now the owner and may enter the critical section.

Thread T1, still in TicketRelease at line 71 then hashes \( k + \text{LongTermThreshold} \) (the sum is 2) to yield index 100 and then increments WaitArray[100] from U to U + 1.

Thread T3 waiting at lines 52-53 observes that change, rechecks Grant, sees that it is close to being granted ownership, exits the long-term waiting loop and switches to classic short-term waiting at lines 60-61. T1 has promoted T3 from long-term to short-term waiting in anticipation of the next unlock operation, to eventually be performed by T2.

Thread T1 now exits the TicketRelease operator.

Thread T2 is the current owner, thread T3 is waiting in short-term mode, and thread T4 is waiting in long-term mode.

## 2 Related Work
Mellor-Crummey and Scott [18] proposed ticket locks with proportional backoff. Waiting threads compare the value of their ticket against the Grant field. The difference reflects the number of intervening threads waiting. That value is then multiplied by some tunable constant, and the thread

Listing 1. Simplified Python-like Implementation of TWA

delays for that period before rechecking grant. The constant is platform- and load-dependent, and requires tuning. In
addition, while the approach decreases the futile polling rate on grant, and may be used in conjunction with polite waiting techniques [10], it does not decrease the invalidation diameter. TWA and ticket locks with proportional backoff both make a distinction among waiting threads based on their relative position in the queue.

Partitioned Ticket Locks [9] augment each ticket lock with a constant-length private array of grant fields, allowing for semi-local waiting. Critically, the array is not shared, and to avoid false sharing within the array, the memory footprint of each lock instance is significantly increased. Ticket Lock “AWN” [?] also uses per-lock array for semi-local waiting. Anderson’s array-based queueing lock [? ?] is also based on ticket locks and employs a per-lock waiting array. As the maximum number of participating threads must be known in advance, the array is sized to ensure purely local spinning.

Various authors [4, 15] have suggested switching adaptively between MCS and ticket locks depending on the contention level. While workable, this adds considerable algorithmic complexity, particularly for the changeover phase, and requires tuning. Lim and Agarwal [16] suggested a more general framework for switching locks at runtime.

3 Empirical Evaluation

Unless otherwise noted, all data was collected on an Oracle X5-2 system. The system has 2 sockets, each populated with an Intel Xeon E5-2699 v3 CPU running at 2.30GHz. Each socket has 18 cores, and each core is 2-way hyperthreaded, yielding 72 logical CPUs in total. The system was running Ubuntu 18.04 with a stock linux version 4.15 kernel, and all software was compiled using the provided GCC version 7.3 toolchain at optimization level “-O3”. 64-bit C or C++ code was used for all experiments. Factory-provided system defaults were used in all cases, and Turbo mode was left enabled. In all cases default free-range unbound threads were used.

We implemented all locks within LD_PRELOAD interposition libraries that expose the standard POSIX pthread_mutex_t programming interface. The framework was made available by the authors of [11]. This allows us to change lock implementations by varying the LD_PRELOAD environment variable and without modifying the application code that uses locks. The C++ std::mutex construct maps directly to pthread_mutex primitives, so interposition works for both C and C++ code. All busy-wait loops used the Intel PAUSE instruction for polite waiting.

We use a 128 byte sector size on Intel processors for alignment to avoid false sharing. The unit of coherence is 64 bytes throughout the cache hierarchy, but 128 bytes is required because of the adjacent cache line prefetch facility where pairs of lines are automatically fetched together.

3.1 MutexBench

The MutexBench benchmark spawns $T$ concurrent threads. Each thread loops as follows: acquire a central lock $L$; execute a critical section; release $L$; execute a non-critical section. At the end of a 10 second measurement interval the benchmark reports the total number of aggregate iterations completed by all the threads. We show the median of 5 independent runs in Figure-2. The critical section advances a C++ std::mt19937 pseudo-random generator (PRNG) 4 steps. The non-critical section uses that same PRNG to compute a value distributed uniformly in [0, 200) and then advances the PRNG that many steps.

As seen in the figure, ticket locks performs the best up to 6 threads, which TWA lagging slightly behind. As we further increase the threads count, however, ticket locks fail to scale. MCS provides stable asymptotic performance that surpasses ticket locks at 24 threads. TWA manages to always outperform MCS, freeing the developer from making a choice between MCS locks and ticket locks.

![Figure 2. MutexBench](image_url)

To show that our approach is general and portable, we next report MutexBench results on a Sun/Oracle T7-2 [7]. The T7-2 has 2 sockets, each socket populated by an M7 SPARC CPU running at 4.13GHz with 32 cores. Each core has 8 logical CPUs sharing 2 pipelines. The system has 512 logical CPUs and was running Solaris 11. 64-bit SPARC does not directly support atomic fetch-and-add or swap operations – these are emulated by means of a 64-bit compare-and-swap operator (CASX). The system uses MOESI cache coherency instead of the MESIF[14] found in modern Intel-branded processors, allowing more graceful handling of write sharing. The graph in Figure-3 has the same shape as found in Figure-2. The abrupt performance drop experienced by all locks starting at 256 threads is caused by competition for pipeline resources.
3.2 throw

The "throw" benchmark launches $T$ threads, each of which loop, executing the following line of C++ code:

```cpp
try { throw 20; } catch (int e) {}.
```

Naively, this construct would be expected to scale linearly, but the C++ runtime implementation acquires mutexes that protect the list of dynamically loaded modules and their exception tables. The problem is long-standing and has proven difficult to fix given the concern that some applications might have come to depend on the serialization [1]. At the end of a 10 second measurement interval the benchmark reports the aggregate number of loops executed by all threads. There is no non-critical section in this benchmark; throw-catch operations are performed back-to back with no intervening delay. In Figure-4 we observe that performance drops significantly between 1 and 2 threads. There is little or no benefit from multiple threads, given that execution is largely serialized, but coherent communication costs are incurred. As we increase beyond two threads performance improves slightly, but never exceeds that observed at one thread. Beyond 2 threads, the shape of the graph recapitulates that seen in MutexBench.

3.3 Random Replacement Cache

The "Random Replacement Cache" benchmark creates a key-value cache with a random replacement policy. All cache operations are protected with a central lock. Both the keys and values are 32-bit integers and we set values equal to a hash of the key. The cache is configured with a capacity limit of 10000 elements. The benchmark launches the specified number of concurrent threads, each of which loops, accessing the cache, and then executing a delay. At the end of a 10 second measurement interval the benchmark reports the aggregate throughput rate. We plot the median of 5 runs for each data point in Figure-5. To emulate locality and key reuse, each thread has a private keyset of 10 recently used keys. We pre-populate the keyset with random keys before the measurement interval, using selection with replacement. With probability $P = 0.9$ a thread picks a random index in its keyset and then uses the corresponding key for its access. We use thread-local C++ `std::mt19937` pseudo-random number generators with a uniform distribution. Otherwise, the thread generates a new random key in the range $[0, 50000)$, installs that key into a random slot in the keyset, and then proceeds to access the cache with that key. The inter-access delay operation picks a random number in the range $[0, 200)$ and then steps the thread-local random number generator that many times.

The cache implementation makes frequent use of malloc-free operations. The default malloc allocator fails to fully scale in this environment and attenuates the benefit conferred by improved locks, so we instead used the index-aware allocator from [2] This allocator uses its own built-in synchronization primitives instead of pthread operators, so LD_PRELOAD interposition on the pthread mutex primitives has no influence on malloc performance.

3.4 `libslock stress_latency`

Figure-6 shows the performance of the "stress latency" benchmark from [8]. The benchmark spawns the specified number of threads, which all run concurrently during a 10 second measurement interval. Each thread iterates as follows: acquire a central lock; execute 200 loops of a delay loop; release the lock; execute 5000 iterations of the same delay loop. The

---

We use the following command line: `./stress_latency -l 1 -d 10000 -a 200 -n <threads> -w 1 -c 1 -p 5000.`
benchmark reports the total number of iterations of the outer loop.

LevelDB uses coarse-grained locking, protecting the database with a single central mutex: DBImpl::Mutex. Profiling indicates contention on that lock via leveldb::DBImpl::Get().

3.5 LevelDB readrandom

In Figure-7 we used the “readrandom” benchmark in LevelDB version 1.20 database varying the number of threads and reporting throughput from the median of 5 runs of 50 second each. Each thread loops, generating random keys and then trying to read the associated value from the database. We first populated a database and then collected data. We made a slight modification to the db_bench benchmarking harness to allow runs with a fixed duration that reported aggregate throughput. Ticket locks exhibit a very slight advantage over MCS and TWA at low threads count after which ticket locks fade and TWA matches or exceeds the performance of MCS.

3.6 LevelDB readwhilewriting

The LevelDB “readwhilewriting” benchmark in Figure-8 spawns $T-1$ random readers (identical to the “readrandom” threads) and a single writer thread which writes to randomly selected keys. The benchmark reports the aggregate throughput completed in a 50 second measurement interval. Each data point is taken as median of 5 distinct runs. The same lock in “readrandom” is the source of contention in this benchmark.

---

8 leveldb.org
9 db_bench --threads=1 --benchmarks=fillseq --db=/tmp/db/
10 db_bench --threads=<threads> --benchmarks=readrandom --use_existing_db=1 --db=/tmp/db/ --duration=50
11 db_bench --benchmarks=readwhilewriting --threads=<threads> --cache_size=50000 --num=100000 --duration=50
3.7 RocksDB readwhilewriting

We next present results in Figure-9 from the RocksDB version 5.14.2 database running their variant of the “readwhilewriting” benchmark. The benchmark is similar to the form found in LevelDB, above, but the underlying database allows more concurrency and avoids the use of a single central lock. We intentionally use a command-line configured to stress the locks that protect the shared LRU cache, causing contention in LRUShard:::lookup().

![Figure 9. RocksDB readwhilewriting](image_url)

4 Conclusion

TWA is a straightforward extension to classic ticket locks, providing the best performance properties of ticket locks and MCS locks. Like ticket locks, it is compact. The key benefit conferred by TWA arises from improved transfer of ownership (handover) in the unlock path, by reducing the number of threads spinning on the grant field at any given time. Even though TWA increases the overall path length in the unlock operation, adding an atomic fetch-and-increment operation, it decreases the effective critical path length for contended handover.

In the Appendix we identify a number of variations on the basic TWA algorithm that we plan to explore in the future.

Acknowledgments

We thank Shady Issa for reviewing drafts and helpful comments.

References

[1] 2016. RFC : introduce dl_iterate_phdr_parallel. https://patchwork.ozlabs.org/patch/652301/
[2] Yehuda Afek, Dave Dice, and Adam Morrison. 2011. Cache Index-aware Memory Allocation. In Proceedings of the International Symposium on Memory Management (ISMM ’11). ACM. https://doi.org/10.1145/1993478.1993486
[3] Vitaly Aksenenov, Dan Alistarh, and Petr Kuznetssov. 2018. Brief Announcement: Performance Prediction for Coarse-Grained Locking. In Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC ’18). ACM. https://doi.org/10.1145/3212734.3212785
[4] Jelena Aštić, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. 2016. Locking Made Easy. In Proceedings of the 17th International Middleware Conference (Middleware ’16). ACM. https://doi.org/10.1145/2988336.2988357
[5] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable locks are dangerous. Ottawa Linux Symposium (OLS) (2012). https://www.kernel.org/doc/ols/docs/2012/ols2012-zeldovich.pdf
[6] Jonathan Corbet. 2014. MCS locks and qspinlocks. https://lwn.net/Articles/590243/
[7] Oracle Corporation. 2014. SPARC T7-2 Server - Oracle Datasheet. http://www.oracle.com/us/products/servers-storage/sparc-t7-2-server-ds-2687049.pdf
[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP ’13). http://doi.acm.org/10.1145/2517349.2522714
[9] David Dice. 2011. Brief Announcement: A Partitioned Ticket Lock. In Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA ’11). http://doi.acm.org/10.1145/1989473.1989543
[10] Dave Dice. 2017. Malthusian Locks. In Proceedings of the Twelfth European Conference on Computer Systems (EuroSys ’17). http://doi.acm.org/10.1145/3064176.3064203
[11] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock Cohorting: A General Technique for Designing NUMA Locks. ACM Trans. Parallel Comput. (2015). https://doi.org/10.1145/2686884
[12] Stijn Eyerman and Lieven Eeckhout. 2010. Modeling Critical Sections in Amdahl’s Law and Its Implications for Multicore Design. In Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA ’10). ACM. https://doi.org/10.1145/1815961.1816011
[13] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. 1979. Resource allocation with immunity to limited process failure. In 20th Annual Symposium on Foundations of Computer Science (FOCS 1979). http://dx.doi.org/10.1109/SFCS.1979.37
[14] J.R. Goodman and H.H.J. Hum. 2009. MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects. https://www.cs.auckland.ac.nz/~goodman/TechnicalReports/MESIF-2009.pdf
[15] P. H. Ha, M. Papatriantafilou, and P. Tsagis. 2005. Reactive spin-locks: a self-tuning approach. In 8th International Symposium on Parallel Architectures,Algorithms and Networks (ISPAN’05). https://doi.org/10.1109/ISPAN.2005.73
[16] Beng-Hong Lim and Anant Agarwal. 1994. Reactive Synchronization Algorithms for Multiprocessors. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI). ACM. https://doi.org/10.1145/195473.195490
[17] O. Krieger B. Rosenburg M. Auslander, D. Edelsohn and R. Wisniewski. 2003. Enhancement to the MCS lock for increased functionality and improved programmability – U.S. patent application number 20030200457. https://patents.google.com/patent/US20030200457

2018-11-09 • Copyright Oracle and its affiliates
We refer to this form as TKT-Dual waits on the global grant Thread T2 (C) state above. Those that are 2 or more elements away from the front have one thread busy-waiting on threads, whereas in our other TWA variants we at most grant 2 threads directly busy-waiting on to take field, at which point it increments the waiting array slot the unlock operator passes ownership to the thread T1 non-trivial set of waiting threads. Incrementing busy-waits on the global grant from the head. This is the immediate successor, and also that is 2 away from the head. This "transitional" thread busy-waits on the waiting array in the usual TWA fashion. (B) The thread of the conceptual queue. These "long-term" threads wait via operator. Briefly, we divide waiting threads into 3 groups: specifically avoiding accesses to the waiting array in unlock promise, yielding results better than that of the baseline of the grant field. Initial experiments with this form show the number of short-term waiters and accelerating handover. May advance the long-term grant field to shift one or more second grant field for long-term waiting. The unlock operator. Two grant fields, one for short-term waiting, for the immediate successor and perhaps a small number of other threads "near" the front of conceptual queue, and a second grant field for long-term waiting. The unlock operator first advances the short-term grant field, and, as needed, may advance the long-term grant field to shift one or more threads from long-term to short-term waiting, constraining the number of short-term waiters and accelerating handover. We refer to this form as TKT-Dual given the dual encoding of the grant field. Initial experiments with this form show promise, yielding results better than that of the baseline ticket lock, although lagging slightly behind TWA.

▶ TWA-Staged We are also exploring variations of TWA where the fast uncontended path for both lock and unlock operations would be identical to that of normal ticket locks, specifically avoiding accesses to the waiting array in unlock operator. Briefly, we divide waiting threads into 3 groups: (A) Those that are 2 or more elements from the front of the conceptual queue. These "long-term" threads wait via the waiting array in the usual TWA fashion. (B) The thread that is 2 away from the head. This "transitional" thread busy-waits on the global grant field. (C) The thread that is 1 away from the head. This is the immediate successor, and also busy-waits on the global grant field. Assume we have a non-trivial set of waiting threads. Incrementing grant in the unlock operator passes ownership to the thread T1 in (C) state above. T1 exits (C) state and becomes the owner. Thread T2 in (B) state also observes the change in the grant field, at which point it increments the waiting array slot associated with the next ticket value – the ticket value one after it’s assigned ticket – to transfer a thread from (A) state to take T2’s place as the (B) thread. T2 then shifts from (B) to (C) state. The downside to this approach is that we have 2 threads directly busy-waiting on grant, the (B) and (C) threads, whereas in our other TWA variants we at most have one thread busy-waiting on grant. The upside is that unlock operator does not access the waiting array, and all the waiting array accesses â€” both loads and fetch-and-add operations â€” are performed by waiting threads. As noted above, the unlock operator simply increments the grant field. This approach leverages those waiting threads to help drive through the (A)→(B)→(C) transitions, reducing the path length of the unlock operation. Relative to classic ticket locks, all changes are encapsulated in the locking slow path.

▶ TWA-ID We note that we can replace the atomic increment of the waiting array element in the unlock path with a simple store of 0 to that location by changing the waiting array elements from counters to unique thread identity references. Threads arriving in the long-term waiting state will write their unique non-zero identity value (which can be as simple as the address of the stack pointer) into the waiting array, recheck the grant value and then busy-wait while the waiting array elements remains equal to the thread identity value they just stored.

We also plan on kernel-level experiments to determine if TWA might be a viable replacement for the Linux kernel’s existing qspinlock construct. Finally, we believe that replacing the waiting array elements with pointers to chains of waiting threads may have benefit. Briefly, each long-term waiting thread would have an on-stack MCS-like queue node that it would push into the appropriate chain in the waiting array, and then use local spinning on a field within that node. Notification of long-term waiters causes the chain to be detached via an atomic SWAP instruction and all the elements are updated to reflect that they should reevaluate the grant field. In the case of collisions, waiting threads may need to re-enqueue on the chain.

5 Appendix : Algorithmic Variations

▶ TKT-Dual An interesting variation on TWA is to forgo the waiting array and simply augment the tick lock structure to use two grant fields, one for short-term waiting, for the immediate successor and perhaps a small number of other threads "near" the front of conceptual queue, and a second grant field for long-term waiting. The unlock operator first advances the short-term grant field, and, as needed, may advance the long-term grant field to shift one or more threads from long-term to short-term waiting, constraining the number of short-term waiters and accelerating handover. We refer to this form as TKT-Dual given the dual encoding of the grant field. Initial experiments with this form show promise, yielding results better than that of the baseline ticket lock, although lagging slightly behind TWA.

▶ TWA-Staged We are also exploring variations of TWA where the fast uncontended path for both lock and unlock operations would be identical to that of normal ticket locks, specifically avoiding accesses to the waiting array in unlock operator. Briefly, we divide waiting threads into 3 groups: (A) Those that are 2 or more elements from the front of the conceptual queue. These "long-term" threads wait via the waiting array in the usual TWA fashion. (B) The thread that is 2 away from the head. This "transitional" thread busy-waits on the global grant field. (C) The thread that is 1 away from the head. This is the immediate successor, and also busy-waits on the global grant field. Assume we have a non-trivial set of waiting threads. Incrementing grant in the unlock operator passes ownership to the thread T1 in (C) state above. T1 exits (C) state and becomes the owner. Thread T2 in (B) state also observes the change in the grant field, at which point it increments the waiting array slot associated with the next ticket value – the ticket value one after it’s assigned ticket – to transfer a thread from (A) state to take T2’s place as the (B) thread. T2 then shifts from (B) to (C) state. The downside to this approach is that we have 2 threads directly busy-waiting on grant, the (B) and (C) threads, whereas in our other TWA variants we at most have one thread busy-waiting on grant. The upside is that unlock operator does not access the waiting array, and all the waiting array accesses â€” both loads and fetch-and-add operations â€” are performed by waiting threads. As noted above, the unlock operator simply increments the grant field. This approach leverages those waiting threads to help drive through the (A)→(B)→(C) transitions, reducing the path length of the unlock operation. Relative to classic ticket locks, all changes are encapsulated in the locking slow path.

▶ TWA-ID We note that we can replace the atomic increment of the waiting array element in the unlock path with a simple store of 0 to that location by changing the waiting array elements from counters to unique thread identity references. Threads arriving in the long-term waiting state will write their unique non-zero identity value (which can be as simple as the address of the stack pointer) into the waiting array, recheck the grant value and then busy-wait while the waiting array elements remains equal to the thread identity value they just stored.

We also plan on kernel-level experiments to determine if TWA might be a viable replacement for the Linux kernel’s existing qspinlock construct. Finally, we believe that replacing the waiting array elements with pointers to chains of waiting threads may have benefit. Briefly, each long-term waiting thread would have an on-stack MCS-like queue node that it would push into the appropriate chain in the waiting array, and then use local spinning on a field within that node. Notification of long-term waiters causes the chain to be detached via an atomic SWAP instruction and all the elements are updated to reflect that they should reevaluate the grant field. In the case of collisions, waiting threads may need to re-enqueue on the chain.

[18] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. ACM Trans. Comput. Syst. (1991). http://doi.acm.org/10.1145/103727.103729
[19] Pedro Ramalhethe and Andreia Correia. 2016. Tidex: A Mutual Exclusion Lock. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’16). http://doi.acm.org/10.1145/2851141.2851171
[20] David P. Reed and Rajendra K. Kanodia. 1979. Synchronization with Eventcounts and Sequencers. Commun. ACM (1979). http://doi.acm.org/10.1145/359060.359076
[21] Michael L. Scott. 2013. Shared-Memory Synchronization. Morgan & Claypool Publishers.
[22] Wikipedia contributors. 2018. Birthday problem — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Birthday_problem&oldid=853622452. [Online; accessed 8-August-2018].

2018-11-09 • Copyright Oracle and or its affiliates