# Reciprocating Locks

Dave Dice
dave.dice@oracle.com
Oracle Labs
USA

Alex Kogan
alex.kogan@oracle.com
Oracle Labs
USA

## ABSTRACT

We present **Reciprocating Locks**, a novel mutual exclusion locking algorithm, targeting cache-coherent shared memory, that enjoys a number of desireable properties. The *doorway* arrival phase and the `Release` operation both run in constant-time. Waitings thread use *local spinning* and only a single *waiting element* is required per thread, regardless of the number of locks a thread might hold at a given time.

While our lock does not provide strict FIFO admission, it still bounds bypass and has strong anti-starvation properties. The lock is compact, space efficient, and has been intentionally designed to be readily usable in real-world general purpose computing environments such as the linux kernel, pthreads, or C++. We show the lock exhibits high throughput under contention and low contention in the uncontended case. The performance of Reciprocating Locks is competitive with and often better than the best state-of-the-art scalable spin locks.

## CCS CONCEPTS

• **Software and its engineering** → *Multithreading*; *Mutual exclusion*; *Concurrency control*; *Process synchronization.*

## KEYWORDS

Synchronization; Locks; Mutual Exclusion; Mutex; Scalability; Cache-coherent Shared Memory

## 1 INTRODUCTION

Locks often have a crucial impact on the performance of parallel software, hence they remain in the focus of intensive research. Many locking algorithms have been proposed over the last several decades.

## 2 THE RECIPROCATING LOCK ALGORITHM

Briefly, under contention, Reciprocating Locks partitions the set of waiting threads into two disjoints lists, which we call the *arrival* and *entry* segments. Threads arriving to acquire the lock will push (prepend) themselves onto a stack, using an atomic exchange operation, forming the *arrival segment*. When the owner releases the lock, it first tries to pass ownership to any threads found in the *entry segment*. Otherwise, if the entry segment is found empty, the thread then uses a atomic exchange to detach the entire current arrival segment (setting it empty), which then becomes the next entry segment, and then passes ownership to the first element of the entry segment.

In Reciprocating Locks, a lock instance consists of an *arrival* word. We deem the lock held if and only the arrival word is non-zero. Specifically, a value of 0 (`nullptr`) encodes the *unlocked* state, 1 encodes the state of *simple locked* – locked with an empty arrival segment – and other values encode being locked, where the remainder of the arrival word points to a stack of threads that have recently arrived at the lock and are waiting for admission, forming the arrival segment. Threads arriving to acquire the lock use an atomic swap (exchange) operator to install the address of a thread-private *waiting element* into the arrival word. If the return value from the atomic exchange was `nullptr` then the arriving thread managed to acquire the lock without contention and can immediately enter the critical section.

Otherwise our thread has encountered contention and must wait. By virtue of the atomic exhange, the thread has managed to push its waiting element onto the arrival segment. The non-`nullptr` value returned from the atomic exchange identifies the next thread in the stack. Similar to the HemLock[21] and CLH[9, 35] lock algorithms, a thread knows only the identity of its immediate neighbor in the arrival segment, and unlike MCS, no explicit linked list of waiting threads is formed or required [1]. That is, the arrival stack is implicit with no `next` pointer fields in the waiting elements. Our thread then proceeds to local spinning on a flag field within its waiting element. This flag will eventually be set during normal ownership succession by some thread running in the `Release` operation, passing ownership to our waiting thread. Our thread, now the owner, then returns the address of the next thread in the entry segment, which was obtained from the atomic exchange. That address, returned from the `Acquire` operation, is passed to the subsequent corresponding invocation of `Release`. The thread identified by that address will subsequently serve as the successor to our current thread. Our thread finally enters the critical section.

In the corresponding `Release` operator, if a successor was passed from the corresponding `Acquire` operator, we simply enable that thread to enter the critical section by setting the flag in its waiting element. Otherwise, we attempt to use an atomic `compare_and_exchange` (CAS) operation to swing the arrival word from *simple locked* state, encoded as 1, back to *unlocked*, encoded as `nullptr`. If the CAS was successful then no waiting threads exist and the lock reverts to *unlocked* state. If the CAS failed, however, additional newly arrived threads must be present on the arrvial segment. In that case we employ an atomic exchange to detach the entire arrival segment, leaving the arrival word in *simple locked* encoded as 1. We then pass lock ownership to the first thread in the detached segment by setting the flag in its waiting element.

Crucially, under contention, threads arrive and join the arrival segment. While the entry segment remains populated, ownership is passed through the entry segment elements in turn. When the

[1] Regarding terminology, say thread $A$ arrives and pushes itself onto the stack and then $B$ follows. In terms of arrival, $B$ is $A$'s *predecessor*. But in terms of subsequent admission order, $B$ is $A$'s *successor*. There is no such situational distinction between arrival and admission for FIFO

current entry segment becomes empty, the `Release` operator detaches the arrival segment, (via an atomic exchange) which then becomes the next entry segment. Threads migrate, in groups, from the arrival segment to the entry segment. The arrival segment consists of those newly arrived threads currently pushed onto the stack anchored at the arrival word while entry segment reflects a set of threads that have already been detached from the arrival stack.

The `Release` operator consults the entry segment first – via the successor reference passed from `Acquire` to `Release` – and passes ownership to the successor if possible. The sequence of successor references passed from `Acquire` to `Release` constitutes the entry segment. But if the entry segment is empty – the passed successor argument is `nullptr` – `Release` then attempts to replenish the entry segment by detaching the arrival segment, and transferring ownership to the first element. In the event the arrival segment is found empty, the lock reverts to *unlocked* state.

The arrival segment is implemented by means of a concurrent *pop-stack*[4], where the key primitives are *push* and *detach-all*, which makes our technique immune to the *A-B-A* pathology [46]. By convention, in Reciprocating Locks, only the current lock holder is allowed to detach the arrival segment.

The waiting element is similar to the CLH or MCS "queue node". In our implementation, we opt to place a thread's wait element in thread-local storage (TLS). As a thread can wait on at most one lock at any given time, such a singleton suffices, and tightly bounds memory usage.

Given that we form a stack for arriving threads, admission order is LIFO within a segment, but remains FIFO between segments. As such, if thread $T1$ pushes itself onto the arrival segment in `Acquire`, and then waits, and $T2$ arrives and pushes itself after $T1$, then a given thread $T2$ can bypass or overtake $T1$ at most once before $T1$ is next granted ownership, providing *thread-specific bounded bypass* and thus avoiding indefinite starvation. Alternatively, we could say Reciprocating Locks provides classic $K$-bounded bypass (worst case) where $K$ reflects the cardinality of the population of threads that might compete for the lock, yielding *population bounded bypass*.

## 3 IMPLEMENTATION DETAILS

```
1  struct WaitingElement {
2  |  std::atomic<int> Gate {0} ;
3  |  std::atomic<WaitingElement *> Terminus {nullptr} ;
4  } ;

5  // Encoding for Lock.Arrivals :
6  // 0:0 = Unlocked
7  // 0:1 = Locked with empty arrival list : SIMPLELOCKED
8  // T:0 = Locked with populated arrival list where T is the
9  //       most recently arrived thread on the arrival stack
10 struct Lock {
11 |  std::atomic <WaitingElement *> Arrivals {nullptr} ;
12 } ;

13 static auto const SIMPLELOCKED = (WaitingElement *) uintptr_t(1) ;
14 |
15 static WaitingElement * Acquire (Lock * L) {
16 |  static thread_local WaitingElement E {} ;
17 |  E.Terminus.store (nullptr , std::memory_order_release) ;
18 |  E.Gate.store   (0      , std::memory_order_release) ;

19 |  auto tail = L->Arrivals.exchange (&E) ;
20 |  assert (tail ≠ &E) ;
21 |  if (tail == nullptr) {
22 |  |  // fast-path uncontended acquire -- We now hold the lock
23 |  |  // Try to replace &E with SIMPLELOCKED.
24 |  |  auto R = L->Arrivals.exchange (SIMPLELOCKED) ;
25 |  |  assert (R ≠ nullptr) ;
26 |  |  if (R == &E) return nullptr ;
27 |  |
28 |  |  // Other threads arrived and pushed onto L->Arrivals in
29 |  |  // the Exchange-Exchange window, above.
30 |  |  // Our &E is now burried at distal end of arrival stack
31 |  |  R->Terminus.store (&E, std::memory_order_release) ;
32 |  |  return R ;
33 |  }
34 |
35 |  // Coerce SIMPLELOCKED to nullptr
36 |  // succ will be our successor when we subsequently release
37 |  auto succ = (WaitingElement *) (uintptr_t(tail) & ~1) ;
38 |  assert (succ ≠ &E) ;
39 |
40 |  // slow path : contention -- waiting phase
41 |  while (E.Gate.load() == 0) Pause() ;
42 |
43 |  // Determine if succession has reached the end of the entry list
44 |  // segment by checking for marker address.
45 |  // E.Terminus == succ implies E is logical terminal WaitingElement.
46 |  // eos is the end-of-segment address
47 |  auto eos = E.Terminus.load() ;
48 |  assert (eos ≠ &E) ;
49 |  if (succ == eos) return nullptr ;
50 |
51 |  // Propagate end-of-segment value through the detached entry list
52 |  // pass successor reference to corresponding Release()
53 |  assert (succ ≠ nullptr && succ->Terminus == nullptr) ;
54 |  succ->Terminus.store (eos, std::memory_order_release) ;
55 |  return succ ;
56 }

57 void Release (Lock * L, WaitingElement * succ) {
58 |  assert (L->Arrivals.load() ≠ nullptr) ;
59 |
60 |  // Case : entry list populated -- appoint successor from entry list
61 |  if (succ ≠ nullptr) {
62 |  |  succ->Gate.store (1, std::memory_order_release) ;
63 |  |  return ;
64 |  }
65 |
66 |  // Case : entry list and arrivals are both empty
67 |  // try fast-path uncontended unlock
68 |  auto v = L->Arrivals.cas (SIMPLELOCKED, nullptr) ;
69 |  if (v == SIMPLELOCKED) return ;
70 |  assert (v ≠ nullptr) ;
71 |
72 |  // Case : entry list is empty and arrivals is populated
73 |  // New threads have arrived and pushed themselves onto the
74 |  // arrival stack.
75 |  // We now detach that segment, shifting those arrivals to
76 |  // become the next entry list segment
77 |  auto w = L->Arrivals.exchange (SIMPLELOCKED) ;
78 |  assert (w ≠ nullptr && w ≠ SIMPLELOCKED) ;
79 |  w->Gate.store (1, std::memory_order_release) ;
80 }
```

**Listing 1: Reciprocating Lock Algorithm**

In Listing-1 we show an implementation of Reciprocating Locks in modern C++. To further explain the operation of Reciprocating Locks we next annotate key scenarios, showing Reciprocating Locks in action.

Our implementation uses conservatively over-fenced C++ `std::atomic` loads and stores for the sake clarity and brevity of explication, even though weaker memory order constraints would suffice and potentially yield better performance. We also assume the existence of a "polite" `Pause()` operator for busy-waiting, and a strong `cas` operator that returns the previous value, which can be trivially implemented via `std::atomic<>::compare_exchange_-strong`.

Finally, for encoding the arrival word, we assume that the low-order bits of wait element addresses are 0.

► **Uncontended** Simple uncontended `Acquire` and `Release` : ❶ Thread $T1$ arrives at Line-15 to acquire lock $L$. $L$'s `arrival` word is currently `nullptr`, indicating that $L$ is in *unlocked* state. ❷ At Line-17 and -18, $T1$ it's thread-specific waiting element, $E$. $T1$ then swaps the address of $E$ into $L1$'s `arrival` word in Line-19. The atomic exchange returns `nullptr`, so control enters the "if" block at Line-22. ❸ At Line-22, $T1$, recognizing there was no contention, tries to replace the address of $E$ in the `arrival` word with the *simple locked* encoding of 1. As no other threads have arrived, the exchange replaces $E$ with 1. ❹ Control returns at Line-26 and the thread can enter and execute the critical section. `Acquire` returns `nullptr` indicating that the entry list is empty and no successor exists. ❺ $T1$ invokes `Release` at Line-57 conveying the `nullptr` value, returned from `Acquire`, through the `succ` argument. ❻ Control reaches Line-68, where $T1$ uses an atomic CAS to try to restore $L$'s `arrival` field from 1 to 0, *unlocked*. As no new threads have arrived, the CAS is successful, and $T1$ returns from `Release` at Line-69.

---

We note that an uncontended `Acquire` requires two atomic exchange operations, at Line-19 and Line-24, increasing the theoretical RMR complexity. In practice, however, as the lock is uncontended, the underlying cache line tends to remain in local *modified* state in $T1$'s cache, assuming normal cache coherent shared memory, so the 2nd exchange incurs very little additional cost. Later, we show a variation, with more complex encodings, that avoids this *double swap* arrival. We also observe that the double swap manifests only absent contention, where the arriving thread found the lock in *unlocked* state. Under sustained contention, we avoid the double swap.

► **Onset of contention** In this scenario we show how to recover from a race in `Acquire` where a thread pushes its wait element onto the stack but, because of other arriving threads, is not able to exchange the arrival word back to 1, and its element becomes "submerged" on the arrival stack. We tolerate this situation by conveying the address of that "submerged" element through the segment, during succession, allowing us to treat the burried element as the effective end-of-segment (equivalent to `nullptr`) and otherwise ignore it. We convey that address through the wait element `Terminus` field, which is normally `nullptr` but will be non-`nullptr` in the case where the race manifested and an element became submerged. In this case we say we have a *zombie* terminal element. During succession a thread checks the address of the successor to determine if matches the submerged terminal element.

❶ Lock $L$ is in *unlocked* state and thread $T1$ invokes `Acquire` on $L$ at Line-15. ❷ $T1$ executes the atomic exchange at Line-19 to install the address of its wait element $E$, which we will designate $E1$, into $L$'s `arrival` word, the exchange returns 0, so $T1$ now holds the lock. ❸ Thread $T2$ now arrives in `Acquire` and exchanges the address of its wait element, $E2$ into the lock's `arrival` word, at Line-19. ❹ Thread $T3$ also arrives in `Acquire` and pushes the address of its wait element, $E3$, onto the arrival stack. ❺ At Line-24, $T1$ attempts to replace the address of its $E1$ with 1, the *simple locked* encoding. The exchange, however, returns $E3$ instead of $E1$, as $T2$ and $T3$ raced $T1$ in the exchange-exchange window (Lines 19 through 24). $T1$ is unable to return at Line-26. At this point, $T1$'s wait element is "burried" or "submerged" in the arrival stack, residing at the distal end, and is not easily removed from the stack. The `arrival` word is 1. $E3$, $E3$'s successor is $E2$, $E2$'s successor is $E1$, and $E1$ has no successor, forming an entry segment that consists of $E3 \rightarrow E2 \rightarrow E1$. Note that $T1$ is the owner, but its own $E1$ also resides on entry segment stack. ❻ To recover, $T1$ installs the address of its own $E1$ into its successor $E3$'s `Terminus` field at Line-31, which is otherwise set to `nullptr` by virtue of Line-17. $T1$ then returns the address of $E3$, at Line-32, to ultimately be used as $T1$'s successor when $T1$ calls `Release`. ❼ $T1$ enters and executes the critcal section. ❽ $T2$ and $T3$, resuming at Line-21, observes that the lock was held and must thus wait, at Line-41. ❾ $T1$ invokes `Release` on $L$. The address $E3$ is passed through the `succ` argument. ❿ $T1$ at Line-62 sets the `Gate` flag in $E3$, passing ownership to $T3$. ⓫ $T1$ returns from `Release` at Line-63. ⓬ $T3$ is now the owner and departs its waiting loop at Line-41, and fetches its `Terminus` field from $E3$ at Line-47, observing the address $E1$. $T3$'s local successor variable (`succ`) refers to $E2$, so the equality check at Line-49 is not true. ⓭ $T3$, at Line-54, passes the address $E1$, which represents the *logical* end-of-segment, into $E2$'s `Terminus` field. ⓮ $T3$, at Line-55, returns $E2$, which is passed to the corresponding `Release`. ⓯ $T3$, returns and executes the critical section. ⓰ $T3$ invokes `Release`, passing $E2$. ⓱ $T3$ at Line-62, sets $E2$'s `Gate` field, passing ownership to $T2$. ⓲ $T3$ returns from `Release` at Line-63. ⓳ $T2$ at Line-41 observes that its `Gate` field was set, indicating that is now the owner of $L$. ⓴ $T2$'s local `succ` variable refers to $E1$. $T2$ fetches from its own $E2$ `Terminus` field, observing $E1$. $T2$, at Line-49, recognizes, via the address-based check, that it has reached the end of the arrival segment, marked by $E1$, and returns `nullptr`. ㉑ $T2$ returns from `Acquire` at Line-49 and enters the critical section. ㉒ $T2$ invokes `Release`, passing `nullptr` as the `succ` argument, indicating the entry segment is empty. ㉓ $T2$ at Line-68, attempts to CAS the `arrival` work from 1 to 0. The CAS succeeds and $L$ is restored to *unlocked* state and $T2$ returns at Line-69. We note that if new threads had arrived and pushed onto the arrival segment, the CAS would fail, and the `Release` operator would detach the arrival stack at Line-77, shifting those arrivals to become the next entry segment, and then pass ownership to the most recently arrived element of the entry segment at Line-79.

---

In practice we find the arrival race between Lines 19 and 24 (the two swaps) to be rare, likely as the window of vulnerability is short, and because it takes time for the coherent interconnect to re-arbitrate the cache line between processors. Also, the race can only occur at the *onset* of contention, when the first arriving thread found the lock not held and then other threads arrived in quick succession. In the event of the race, at Line-31, we pass the address of $E$ through the `Terminus` field. We are using $E$'s address for addressed-based comparisons (Line-49) as a distinguished marker or sentinel to indicate the logical end-of-segment. $E$ itself, however, will not be subsequently accessed by succession within the segment. We note that elements associated with a given thread can appear on at most

one segment at any time, but, when an address is used as an end-of-segment marker, it is possible that it appears on both the arrival segment and entry segment.

We observe that the `Terminus` field could also reside in the lock body instead of in the waiting elements. While viable, that approach increases the size of the lock body, and increases induced coherence traffic. Instead, we borrow a technique from Compact NUMA-Aware Locks (CNA) [15] and avoid such shared central fields by propagating information – in this case the address of the terminal element of the segment – through the chain of waiting elements.

► **Sustained contention** ❶ Lock $L$ is initially in *unlocked* state. Thread $T1$ arrives in `Acquire` at Line-15. ❷ $T1$'s exchange operation at Line-19 installs the address of $T1$'s wait element, $E1$, into $L$'s `arrival` word. As the exchange returned `nullptr`, $T1$ has acquired the lock. $T1$ then, via the exchange at Line-24, replaces $E1$ with 1 and returns `nullptr` at Line-26. ❸ $T1$ enters and executes the critical section. ❹ While $T1$ holds $L$, thread $T2$ arrives $E2$ pushes onto the arrival stack. The exchange operation at Line-19 returns 1 into $T2$'s local `tail` variable. As `tail` is non-0, $T2$ must wait and control passes through Line-37, which coerces the value 1 to `nullptr`, as there are no successors in the arrival segment. We interpret 1 (simple locked) as effectively equivalent to `nullptr` for the purposes of forming the arrival segment. $T2$ waits on $E2$ at Line-41. ❺ With $T1$ still holding $L$, $T3$ also arrives and uses the atomic exchange to push its element $E3$ onto the arrival stack. The exchange returns $E2$ and Line-37 leaves $E2$ unchanged. $T3$'s `succ` variable points to $E2$. The arrival stack consists of $E3$ followed by $E2$. The arrival word points to $E3$ and $T3$'s `succ` variable points to $E2$, while $E2$'s `succ` variable is `nullptr`, indicating that $E2$ is the final element on the arrival segment. The entry segment is empty. $T3$ waits on $E3$ at Line-41. ❻ Similarly, $T4$ arrives and pushes $E4$ onto the arrival stack and then waits. ❼ $T1$ eventually calls `Release`. As the `succ` argument is `nullptr`, indicating an empty entry segment, $T1$ then attempts the CAS, which fails. $T1$ then executes exchange(1) to detach the arrival segment at Line-77. The exchange operator returns $E4$. The arrival segment is now empty and the entry segment consists of $E4$ then $E3$ then $E2$. $T1$ passes ownership to $T4$ at Line-79. ❽ $T4$ departs its waiting phase at Line-41. The `Terminus` value is `nullptr` and does not equal `succ`, which is $E3$, so control passes through Line-52, which performs a redundant but benign store of 0 of 0. $T4$ returns $E3$ as its successor at Line-55, and then enters the critical section. The arrival segment is currently empty and the entry segment consists of just $E3$ and $E2$. ❾ while $T3$ holds $L$, $T5$ and then $T6$ arrive to acquire $L$, pushing $E5$ and then $E6$, respectively, onto the arrival stack. The arrival segment consists of $E6$ then $E5$ and the detached entry segment is just $E3$ and $E2$. ❿ $T4$ calls `Release`, passing $E3$ as the successor, and we grant ownership to $T3$ at Line-62. ⓫ $T3$ departs its wait phase at Line-41 and returns $E2$ as its successor at Line-55. ⓬ $T3$ enters and excutes the critical section. ⓭ $T3$ calls `Release` passing $E2$ as its successor and grants ownership to $T2$ at Line-62. ⓮ $T2$ departs its wait phase at line-41 and returns `nullptr` as its successor at Line-49, as we have reached the end of the entry segment. ⓯ $T2$ enters and executes the critical section. ⓰ $T2$ invokes `Release` passing `succ=nullptr`. As the entry segment is now empty and there is no immediate successor, we bypass the block at Lines 61-62 and then attempt the CAS at Line-68, which fails, as the arrival segment is populated. $T2$ then detaches the arrival segment of $E6$ then $E5$ at Line-77, leaving the `arrival` word set to 1, and passes ownership to $T6$ at Line-79. ⓱ $T6$ exits its waiting loop at Line-41 and returns $E5$ as its successor. ⓲ $T6$ enters and executes the critical section. ⓳ $T6$ invokes `Release` passing `succ=E5` and we grant ownership to $T5$ at Line-62. ⓴ $T5$ exits its waiting loop at Line-39 and returns `nullptr` as its successor ㉑ $T5$ enters and executes the critical section. ㉒ $T5$ calls `Release` passing `succ=nullptr`. Both the entry segment and arrival segment are empty. The CAS at Line-68 succeeds, and the $L$ is restored to *unlocked* state.

---

As described above, we use a non-standard (non-pthreads) locking interface, where the identity of the owner's successor on the entry segment – embodied as the address of the successor thread's waiting element – is returned and passed from the `Acquire` operation to the corresponding `Release` operator. To avoid imposing a non-standard interface, and provide a standard *context free* programming interface [48], we can slightly modify the implementation to pass the address of the successor through an extra field in the lock body, which can induce extra coherence traffic, or keep track of held locks in thread-local storage (TLS) and convey the information in that fashion. We note that MCS and CLH also also must pass additional contextual information from `Acquire` to `Release`, so this concern is not specific to Reciprocating Locks. Most locking algorithms that are not innately *context free* can be transformed to become *context free* through such techniques. In all lock implementations used in the benchmark section of this paper, we elected to pass any additional context information via extra fields in the lock instance.

Modern C++ locking constructs such as `std::scoped_lock` and `std::lock_guard`, by means of the *Resource Acquisition is Initialization(RAII)*[51] idiom, where the constructor acquires the lock and the destructor releases the lock, manage to avoid explicit `lock` and `unlock` calls in application code. This same design pattern readily supports underlying lock primitives that require context to be passed. Specifically, we can pass pass extra context through additional fields in the wrapper classes. Likewise, locking interfaces that specify the critical section as a C++ lambda also allow such latitude. Interfaces that use scoped locking, such as Java's "synchronized" construct, permit the implementation to trivially pass information from the underlying `lock` cite to the corresponding `unlock`.

We assume the existence of an wait-free atomic exchange operator. Specifically, the implementation thereof should *not* be via loops that employ optimistic `compare-and-swap` or `load-locked(LL) and store-conditional(SC)` primitives. In particular we assume that C++ `atomic<> exchange and compare_and_exchange` primitives are implememented in a wait-free fashion, as is the case on AMD or Intel x86 processors or ARM processors that support the LSE instruction subset.

One possible disadvantage to Reciprocating Locks is that we might be required to access the central shared lock body more than, say, under MCS. MCS updates the lock's head pointer as threads arrive and use exchange to join the queue of waiting threads. But if

contention is steady-state and sustained, all hand-over of ownership in the `Release` operations can be accomplished directly thread-to-thread without needing to access the lock body. In Reciprocating Locks, however, we need to periodically consult the arrival stack to replenish the entry segment, potentially increasing RMR (remote memory reference) complexity and generating additional coherence traffic. Interestingly, as more thread contend, Reciprocating Locks needs to detach the arrival segment less often, somewhat mitigating the concern.

## 4 ADDITIONAL REQUIREMENTS

We target lock algorithms that are suitable for environments such as the Linux kernel, as a replacment for the user-level `pthreads_mutex` primitive, or for use in runtime environments such as the Java Virtual Machine (JVM). As such, we indentify additional de-facto requirements for general purpose lock algorithms.

**Safe Against Prompt Lock Destruction** Various lock algorithms perform stores in the `Release` operation that potentially release ownership, but then perform addition accesses to fields in the lock body to ensure succession and progress. This can result in a class of unsafe use-after-free memory reference errors [7, 39]. A detailed description of such an error can be found in [21]. All algorithms for use in the linux kernel or in the pthreads environment are expected to be prompt lock destruction safe.

**Support for Large Numbers of Extant Threads** The algorithms must support the simultaneous participation of an arbitrary number of threads, where threads are created and destroyed dynamically, and the peak number of extant threads is not known in advance.

**Support for Large Numbers of Extant Locks** Similarly, the algorithm needs to support large numbers of lock instances, which can be created and destroyed dynamically. Recent work [32] shows the Linux kernel has more than 6000 statically initialized lock instances. Again, the number of locks is not know in advance, as drivers load, and unload, for instance, or as data structures dynamically resize.

**Plural Locking** A given thread is expected to be able lock and hold a large number of locks simulatenously. In the Linux kernel, for instance, situations arise when 40 or more distinct locks are held by a single thread at a given time, as evidenced by the `MAX_LOCK_DEPTH` tunable, which is used by the kernel's "lockdep" facility to track the set of locks held in an explicit per-thread list for the purposes of detecting potential deadlock. Furthermore, locks must be able to be released in non-LIFO imbalanced order as it is fairly common to acquire a lock in one routine, return, and then release the lock in the caller.

**Space Efficient** We expect the lock algorithm to be frugal and parsimonious with regard to space usage. Critically, any algorithms that require per-lock and per-thread storage – with $Threads * Locks$ space consumption – such as Anderson's Array-Based Queue Lock [3], are not suitable.

**FIFO as a non-goal** We also note that many real-world lock implementations, such as the current implementation of Java's "synchronzied" in the HotSpot Java Virtual Machine, the `java.util.Concurrent.ReentrantLock`, and the default `pthread_mutex` on Linux, are non-FIFO, and in fact permit unbounded bypass and indefinite starvation. Relatedly, Compact NUMA-Aware Locks (CNA) [15] intentionally imposes non-FIFO admission order to improve throughput on NUMA platforms, trading aggregate throughput against strict FIFO ordering [2]. In fact, strict FIFO locking is an anti-pattern for common practical lock designs, as it suffers from reduced throughput compared to more relaxed admission schedules, and has shortcomings when used with waiting techniques that deschedule threads, or when involuntary preemption is in play [10, 13, 17, 18]. Non-FIFO admission schedules allow more opportunism that translates into perforamnce.

---

Given these particular constraints, we exclude a number of algorithms from consideration. Dvir's algorithms [24], Rhee's algorithm [43], Lee's HL1 and HL2 [30, 31], and Jayanati's "Ideal Queued Spinlock" [28] [29] algorithms, when simplified for use in cache-coherent (CC) environments, all have extremely simple elegant paths and minimal RMR complexity, suggesting they would be competitive and good candidates, but they do not readily tolerate multiple locks being held simultaneously. The above tend to use so-called "node-toggling" and "node-switching" techniques – also called "two-face" by Lee – that, while they work well when a thread holds at most one lock, are awkward for general purpose use. Our specific concerns are space blow-up the need to maintain toggle element pairs and a "face" index for pairs of locks and threads, in addition to the requirement that we re-associate that metadata with the lock instance at release-time. Some of Dvir and Lee's algorithms are also not safe against prompt lock destruction.

## 5 RELATED WORK

While mutual exclusion remains an active research topic [1, 9, 11–17, 23, 24, 27–29, 38, 41, 44] we focus on locks closely related to our design.

Simple test-and-set or polite test-and-test-and-set [44] locks are compact and exhibit excellent latency for uncontended operations, but fail to scale and may allow unfairness and even indefinite starvation. Ticket Locks are compact and FIFO and also have excellent latency for uncontended operations but they also fail to scale because of global spinning, although some variations attempt to overcome this obstacle, at the cost of increased space [16, 20, 40]. For instance Anderson's array-based queueing lock [2, 3] is based on Ticket Locks but provides local spinning. It employs a waiting array for each lock instance, sized to ensure there is at least one array element for each potentially waiting thread, yielding a potentially large footprint. The maximum number of participating threads must be known in advance when initializing the lock. TWA[16] is a variation on ticket locks that reduces the incidence of global spinning.

Queue-based locks such as MCS or CLH are FIFO and provide local spinning and are thus more scalable. MCS is used in the linux kernel for the low-level "qspinlock" construct [6, 8, 33]. Modern extensions of MCS edit the queue order to make the lock *NUMA-Aware*[15]. MCS readily allows editing and re-ordering of the queue

[2]We note in passing that even if a lock has strict FIFO admission order, otherwise identical threads can still stuffer from persistent long-term unfairness related to shared cache residency imbalance, reflecting the "Matthew Effect" [19].

of waiting threads, [11, 15, 36] whereas editing the chain is more difficult under CLH, HemLock and Reciprocating Locks, where there no explicit linked lists.

CLH is extremely simple, has minimal RMR complexity, and requires just an single atomic exchange operation in the `Acquire` operation and no atomic read-modify-write instructions in `Release`. Unfortunately the waiting elements migrate between threads, which is inimical to performance in NUMA enviroments. CLH locks also require explicit constructors and destructors, which may be inconvenient [3] We use a variation on Scott's [44] Figure 4.14, which converts the CLH lock to be *context free* albeit at the cost of adding an extra field to the lock body to convey the address of the head waiting element to `Release`.

The K42 [34, 44] variation of MCS can recover the queue element before returning from `Acquire` whereas classic MCS recovers the queue element in `Release`. That is, under K42, a queue element is needed only while waiting but not while the lock is held, and as such, queue elements can always be allocated on stack, if desired. While appealing, the paths are much more complex and touch more cache lines than the classic version, impacting performance. In addition, neither the doorway nor the `Release` path operate in constant time.

HemLock combines aspects of both CLH and MCS to form a lock that has very simple waiting node lifecycle constraints, is *context free* but still scales well. HemLock does not provide constant remote memory reference (RMR) complexity [24]. Similar to MCS, HemLock lacks a completely constant-time unlock operation, whereas the unlock operator for CLH and Tickets is constant-time. Unlike MCS, HemLock requires active synchronous back-and-forth communication in the unlock path between the outgoing thread and its successor to protect the lifecycle of the waiting element. We note, however, that HemLock remains constant-time in the `Release` operator to the point where ownership is conveyed to the successor. HemLock uses *address-based* transfer of ownership, writing the address of the lock instead of a boolean, differentiating it from MCS and CLH. Reciprocating Locks, like HemLock, requires just a singleton per-thread waiting element allocated in thread-local storage.

For both HemLock and Reciprocating Locks, the amount of memory required for locking is $T * W + L * B$ where $T$ is the number of extant threads, $W$ is the size of the waiting element in thread-local storage, $L$ is the number of currently extant locks, and $B$ is the size of the lock body.

In Table-1 we compare the attributes of various local algorithms.

**Local Spinning**: reflects whether the lock utilizes *local*, *global* or *semi-local* spinning. **Contant-time Doorway**: denotes whether the lock `Acquire` operator has a constant-time arrival "doorway". **Constant-time Unlock**: indicates if the lock `Release` operation is bounded. **FIFO**: does the lock provide strict FCFS admission. Note-1 HemLock, for instance, is lock-free up point where the lock is either released or transferred to a successor, but the `Release` operator, for reasons of memory safety, then waits for the successor to acknowledge transfer of ownership before the memory underlying the queue element can be potentially reused. Specifically, in HemLock an uncontended `Release` operation is contant-time and a contended `Release` is constant-time up to and including the point where ownership is conveyed to the successor. **Context-free**: indicates additional information does not need to be transfered from the lock operator to the corresponding `Release` operation. **Complexity - `Acquire`** and **Complexity - `Release`** : we measured the size of the `Acquire` and `Release` methods in units of platform independent LLVM intermediate representation (IR) instructions, as emitted by `clang++-18`, which serves as a proxy for complexity. We note that much of the complexity found in TWA manifests through the use of the hash function which maps lock address and ticket value pairs to slots in the waiting array. **On-Stack**: indicates the queue elements, if any, may be allocated on-stack. This also implies the nodes do not migrate and have a tenure constrained to the duration of the locking episode. **Nodes Circulate**: queue elements migrate between threads. This often implies the need for an explicit queue element lifecycle management system and precludes convenient on-stack allocation of queue elements. Migration may also be unfriendly to performance in NUMA environments. **Explicit CTOR/DTOR Required**: indicates the lock requires non-trivial constructors or destructors. CLH, for instance, requires destructors to run to release the wait elements referenced in the lock, to avoid memory leaks.

[3] Many lock implementations require just trival constructors to set the lock fields to 0 or some constant, and trivial destuctors, which do nothing. The Linux kernel `spinlock_t`/`qspinlock_t` interface provides a constructor, but does not even expose a destructor. Similarly, C++ `std::mutex` is allowed to be *trivially destructable*, meaning storage occupied by trivially destructible objects may be reused without calling the destructor. Under both GCC g++ version 13 and Clang++ Version 18, `is_trivially_destructible` reports `True` for `std::mutex`, and as such, destructors do not run.

## 6 EMPIRICAL RESULTS

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 20.04 with a stock Linux version 5.4 kernel, and all software was compiled using the provided GCC version 9.3 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 [47] was left enabled. In all cases default free-range unbound threads were used (no pinning of threads to processors).

We implemented all user-mode locks within LD_PRELOAD interposition libraries that expose the standard POSIX `pthread_mutex_t` programming interface using the framework from [23]. 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 lock busy-wait loops used the Intel `PAUSE` instruction. To reduce false sharing, all lock instances and waiting elements were aligned and sequestered at 128-byte boundaries.

### 6.1 MutexBench benchmark

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

| Property | Lock Algorithm | | | | | |
|---|---|---|---|---|---|---|
| | MCS | CLH | HemLock | Ticket | TWA | Reciprocating |
| Spinning | Local | Local | Semi | Global | Semi | Local |
| Constant-time Doorway | Yes | Yes | Yes | Yes | Yes | Yes |
| Constant-time `Release` | No | | Note-1 | | | |
| Context-free | No | No | | | | No |
| FIFO | | | | | | No |
| Complexity – `Acquire` | 29 | 19 | 22 | 12 | 57 | 42 |
| Complexity – `Release` | 17 | 4 | 21 | 5 | 11 | 20 |
| On-Stack | | No | No | N/A | N/A | No |
| Nodes Circulate | | Yes | | N/A | N/A | |
| Explicit CTOR/DTOR Required | | Yes | | | | |

**Table 1: Comparison of lock algorithm properties**

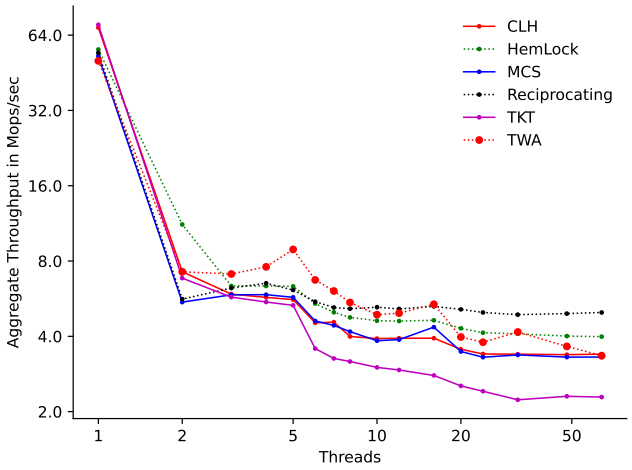


**Figure 1: MutexBench : Maximum Contention**

10 second measurement interval the benchmark reports the total number of aggregate iterations completed by all the threads. We report the median of 7 independent runs in Figure-1 where the critical section is empty as well as the non-critical section, subjecting the lock to extreme contention. (At just one thread, this configuration also constitutes a useful benchmark for uncontended latency). The $X$-axis reflects the number of concurrently executing threads contending for the lock, and the $Y$ reports aggregate throughput. For clarity and to convey the maximum amount of information to allow a comparison of the algorithms, the $X$-axis is offset to the minimum score and the $Y$-axis is logarithmic.

We ran the benchmark under the following lock algorithms: **MCS** is classic MCS. To avoid memory allocation during the measurement interval, the MCS implementation uses a thread-local stack of free queue elements. **CLH** is CLH based on Scott's *CLH variant with a standard interface* Figure-4.14 of [44]; For the MCS and CLH locks, our implementation stores the current head of the queue – the owner – in a field adjacent to the tail, so the lock body size was 2 words. CLH presents something of a challenge when used under the `pthread_mutex` interface First, pthreads allows the programmer to use trivial initializers – setting the mutex body to all 0 – and avoid calling `pthread_mutex_init`. To compensate, we modified `pthread_mutex_lock` to populate such an initialized lock with the CLH "dummy node" lazily, on-demand, on the first lock operation. In `pthread_mutex_destroy` we free the node, if populated, but many pplications also do not call `pthread_mutex_destroy`, which constitutes a memory leak. As such, we avoided applications that create and then abandon large numbers of locks in a dynamic fashion. The Ticket Lock also has a size of 2 words, while HemLock requires a lock body of just 1 word. MCS and CLH additionally require one queue element for each lock held or waited upon. **Ticket** is a classic Ticket Lock; **HemLock** is the HemLock algorithm, with the CTR optimization.

In Figure-1 we make the following observations regarding operation at maximal contention with an empty critical section: [4].

- At 1 thread the benchmark measures the latency of uncontended `Acquire` and `Release` operations. Ticket Locks are the fastest, followed closely by HemLock, Reciprocating Locks, CLH and MCS.
- As we increase the number of threads, Ticket Locks initially do well but then fade, exhibiting a precipitous drop in performance. TWA is the clear leader in the "middle" area of the graph, between 4 and 16 threads.

[4]We note in passing that care must be taken when *negative* or *retrograde* scaling occurs and aggregate performance degrades as we increase threads. As a thought experiment, if a hypothetical lock implementation were to introduce additional synthetic delays outside the critical path, aggregate performance might increase as the delay throttles the arrival rate and concurrency over the contended lock [26]. As such, evaluating just the maximal contention case in isolation is insufficient.

- Broadly, at higher thread counts, HemLock performs slightly better than or the same as CLH or MCS, while Reciprocating Locks provides the best throughput.

We opted to exclude NUMA-aware locks such as Cohort Locks [22, 23] and Compact NUMA-Aware Locks (CNA) [12, 15] from consideration.

## 7 FUTURE WORK

We plan on exploring the "coherence traffic reduction" optimization (CTR), from HemLock, with Reciprocating Locks.

## 8 CONCLUSION

Reciprocating Locks is the first lock algorithm to have a fully constant-time doorway phase and a contant-time `Release` but which is still practical and fulfills the criteria for general purpose locking [5].

## REFERENCES


[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. *SIGPLAN Notices OOPSLA 1999*, 1999. doi:10.1145/320385.320402.

[2] J.H. Anderson, Y.J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 2003. URL: https://doi.org/10.1007/s00446-003-0088-6.

[3] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1990. doi:10.1109/71.80120.

[4] D. Avis and M. Newborn. On pop-stacks in series. *Utilitas Math. 19*, pages 129–140, 1981.

[5] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[6] Jonathan Corbet. MCS locks and qspinlocks. https://lwn.net/Articles/590243, March 11, 2014. Accessed: 2018-09-12.

[7] Jonathan Corbet. A surprise with mutexes and reference counts. https://lwn.net/Articles/575460, December 4, 2013.

[8] Jonathan Corbet. Mcs locks and qspinlocks, 2014. URL: https://lwn.net/Articles/590243/.

[9] Travis Craig. Building fifo and priority-queueing spin locks from atomic swap, 1993.

[10] Dave Dice. Malthusian locks. *CoRR*, abs/1511.06035, 2015. URL: http://arxiv.org/abs/1511.06035, arXiv:1511.06035.

[11] Dave Dice. Malthusian locks. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017. URL: http://doi.acm.org/10.1145/3064176.3064203.

[12] Dave Dice and Alex Kogan. Compact numa-aware locks. *CoRR*, abs/1810.05600, 2018. URL: http://arxiv.org/abs/1810.05600.

[13] Dave Dice and Alex Kogan. Avoiding scalability collapse by restricting concurrency. In *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*, Lecture Notes in Computer Science. Springer, 2019. doi:10.1007/978-3-030-29400-7_26.

[14] Dave Dice and Alex Kogan. Avoiding scalability collapse by restricting concurrency. *CoRR*, abs/1905.10818, 2019. URL: http://arxiv.org/abs/1905.10818, arXiv:1905.10818.


[5] The name *Reciprocating Locks* arises from the following analogy with a piston-based compressor or reciprocating pump. The pump's cylinder has distinct intake and exhaust ports and the piston is initially positioned at the bottom of the cylinder. Threads intending to acquire the lock – say, $A$, $B$ and $C$ where $A$ was the first to arrive and $C$ last – arrive at the intake port and wait there. On the intake cycle, the piston pulls those threads from the intake port into the cylinder body. On the subsequent exhaust cycle, the piston expels the threads through the exhaust port. Passing through the exhaust port is analogous to passing through the lock's critical section. Like packets of air or gas, the threads are expelled from the cylinder through the exhaust port in an order reversed from that in which they originally arrived : $C$ then $B$ then $A$. The *entry segment* corresponds to the current cylinder contents and the *arrival segment* reflects threads that reside in the intake manifold, before entering the cylinder. The non-critical section is where threads circulate back from the exhaust to the intake. The intake phase is analogous to detaching the current arrival segment and shifting those threads into the entry segment.


[15] Dave Dice and Alex Kogan. Compact numa-aware locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. Association for Computing Machinery, 2019. doi:10.1145/3302424.3303984.

[16] Dave Dice and Alex Kogan. TWA - ticket locks augmented with a waiting array. In *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*. Springer, 2019. doi:10.1007/978-3-030-29400-7_24.

[17] Dave Dice and Alex Kogan. Fissile locks, 2020. URL: https://arxiv.org/abs/2003.05025, arXiv:2003.05025.

[18] Dave Dice and Alex Kogan. Fissile locks. In *Networked Systems (NETYS 2020)*, 2021. URL: https://doi.org/10.1007/978-3-030-67087-0_13.

[19] Dave Dice, Virendra J. Marathe, and Nir Shavit. Persistent unfairness arising from cache residency imbalance. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, 2014. URL: https://doi.org/10.1145/2612669.2612703.

[20] David Dice. Brief announcement: A partitioned ticket lock. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, 2011. URL: http://doi.acm.org/10.1145/1989493.1989543.

[21] David Dice and Alex Kogan. Hemlock : Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2021. URL: https://doi.org/10.1145/3409964.3461805.

[22] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12. Association for Computing Machinery, 2012. doi:10.1145/2145816.2145848.

[23] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 2015. URL: http://doi.acm.org/10.1145/2686884, doi:10.1145/2686884.

[24] Rotem Dvir and Gadi Taubenfeld. Mutual Exclusion Algorithms with Constant RMR Complexity and Wait-Free Exit Code. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8652, doi:10.4230/LIPIcs.OPODIS.2017.17.

[25] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *20th Annual Symposium on Foundations of Computer Science (FOCS 1979)*, 1979. URL: http://dx.doi.org/10.1109/SFCS.1979.37.

[26] Carlos Gershenson and Dirk Helbing. When Slower is Faster. *CoRR*, 2011. URL: http://arxiv.org/abs/1506.06796v2.

[27] Wim H. Hesselink and Peter A. Buhr. Mcsh, a lock with the standard interface. *ACM Trans. Parallel Comput.*, 2023. URL: https://doi.org/10.1145/3584696.

[28] Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Towards an ideal queue lock. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, ICDCN 2020. Association for Computing Machinery, 2020. URL: https://doi.org/10.1145/3369740.3369784.

[29] Siddhartha Visveswara Jayanti. Simple, fast, scalable, and reliable multiprocessor algorithms, 2023.

[30] Hyonho Lee. Local-spin mutual exclusion algorithms on the dsm model using fetch&store objects. Masters Thesis, University of Toronto. URL: http://www.cs.toronto.edu/pub/hlee/thesis.ps.

[31] Hyonho Lee. Transformations of mutual exclusion algorithms from the cache-coherent model to the distributed shared memory model. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, 2005. doi:10.1109/ICDCS.2005.83.

[32] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. Association for Computing Machinery, 2019. doi:10.1145/3302424.3303948.

[33] Waiman Long. qspinlock: Introducing a 4-byte queue spinlock implementation. https://lwn.net/Articles/561775, July 31, 2013, 2013. Accessed: 2018-09-19.

[34] O. Krieger B. Rosenburg M. Auslander, D. Edelsohn and R. Wisniewski. Enhancement to the mcs lock for increased functionality and improved programmability – u.s. patent application number 20030200457, 2003. URL: https://patents.google.com/patent/US20030200457.

[35] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, 1994. doi:10.1109/IPPS.1994.288305.

[36] Evangelos P. Markatos and Thomas J. LeBlanc. Multiprocessor synchronization primitives with priorities. 8th IEEE Workshop on Real-Time Operating Systems and Software. IEEE, 1991.

[37] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 1991. URL: http://doi.acm.org/10.1145/103727.103729.

[38] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91. ACM, 1991. URL: http://doi.acm.org/10.1145/109625.109637.

[39] Atsushi Nemoto. Bug 13690 – pthread_mutex_unlock potentially cause invalid access. https://sourceware.org/bugzilla/show_bug.cgi?id=13690, February 14, 2012.
[40] Pedro Ramalhete. Ticket lock - array of waiting nodes (awn), 2015. URL: http://concurrencyfreaks.blogspot.com/2015/01/ticket-lock-array-of-waiting-nodes-awn.html.
[41] Pedro Ramalhete and Andreia Correia. Tidex: A mutual exclusion lock. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016. URL: http://doi.acm.org/10.1145/2851141.2851171.
[42] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 1979. URL: http://doi.acm.org/10.1145/359060.359076.
[43] I. Rhee. Optimizing a fifo, scalable spin lock using consistent memory. In *17th IEEE Real-Time Systems Symposium*, 1996. doi:10.1109/REAL.1996.563705.
[44] Michael L. Scott and Trevor Brown. *Shared-Memory Synchronization, Second Edition*. Springer, 2024. doi:10.1007/978-3-031-38684-8.
[45] Harold S. Stone and Dominique Thibaut. Footprints in the cache. *SIGMETRICS Perform. Eval. Rev.*, 1986. doi:10.1145/317531.317533.
[46] R.K. Treiber, 1986.
[47] U. Verner, A. Mendelson, and A. Schuster. Extending amdahl's law for multicores with turbo boost. *IEEE Computer Architecture Letters*, 2017. URL: https://doi.org/10.1109/LCA.2015.2512982.
[48] Tianzheng Wang, Milind Chabbi, and Hideaki Kimura. Be my guest: Mcs lock now welcomes guests. *SIGPLAN PPoPP*, 2016. doi:10.1145/3016078.2851160.
[49] Wikipedia. Cocktail shaker sort, 2018. URL: https://en.wikipedia.org/wiki/Cocktail_shaker_sort.
[50] Wikipedia. Gnome sort, 2018. URL: https://en.wikipedia.org/wiki/Gnome_sort.
[51] Wikipedia. Resource acquisition is initialization, 2022. URL: https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization.
[52] Liang Yuan, Chen Ding, Wesley Smith, Peter Denning, and Yunquan Zhang. A relational theory of locality. *TACO – ACM Trans. Archit. Code Optim.*, 2019. doi:10.1145/3341109.

# A PALINDROMIC ADMISSION SCHEDULES

Reciprocating Locks may allow a *palindromic* admission schedule which, under the right circumstances, can persist for long periods.

## A.1 Example Scenario

Table-2 illustrates the phenomena with a simple scenario. Threads *A B C* and *D* all complete for a given lock *L*. Initially, at time 1, *A* is the owner, executing in the critical section, the entry segment consists of *B* then *D* and the arrival segment contains *D*. The non-critical section is empty, so when a thread releases the lock, it immediately tries to reacquire. *A* completes the critical section and invokes `Release`, which passes ownership to the head of the entry segment, *B*, and *A* recirculates, calls `Acquire` again, and joints the arrival segment, reflecting the state at time 2. Next, *B*, releases the lock, and passes ownership to *C*, leaving the entry segment empty. *B* then prepends itself to the arrival segment stack, which now contains *B-A-D*, as shown at time 3. *C* then releases the lock. As the entry segment is empty, *C* reverts to and detaches the entire arrival segment, migrating that set of threads en-masse to become the next entry segment. *C* then picks *B* from the new entry segment, leaving *A* and *D* in the entry segment, and transfers ownership to *B*. *C* then tries to reacquire and emplaces itself on the arrival segment, leaving the configuration as seen at time 4. *B* completes the critical section and then releases the lock, shifting ownership to *A*, and then pushes itself onto the arrival segment, reflecting the state at time 5. *A* finishes the critical section and releases the lock, passing ownership to *D*. *A* then recirculates, calls `Acquire`, and pushes itself into the front of the arrival segment, leaving the state as seen at time 6. *D* releases the lock, detaches the arrival segment to become the next entry segment, and passes ownership to *A*, leaving the configuration as seen at time 7.

## A.2 Long-term Admission Unfairness

As we can see, the state at time 7 is the same as that at time 1, so the admission sequence, $A-B-C-B-A-D$, repeats, having a period length of 6 steps. While there is no long-term starvation, within that cycle we see that *A* and *B* were admitted two times, while *D* and *C* were admitted just once, which can manifest as long-term relative unfairess between the participating threads. While not a perfect palindrome, we say such a schedule is *palindromic* [6]. We claim the worst case unfairness that might manifest is $2X$.

## A.3 Cache-residency Unfairness

We identify another distinct source of long-term unfairness that can arise from palindromic admissions, above and beyond the simple issue, above, of some threads being admitted less frequently. We assume that all circulating threads over a lock access the same shared last-level cache (LLC). While threads are waiting, their residency in the LLC undergoes exponential decay because of the actions of the other threads executing in the critical section or their respective non-critical sections. Considering a true repeating palindome

[6] or *boustrophedonic*, which is also somewhat akin to an *elevator seek* ordering, where the elevator visits the extreme floors at a lower unit time rate than the interior floors. Similarly, if two back-to-back passes need to be made over an array or linked list, a well-known optimization is to run the second pass in reverse to improve cache residency [49, 50].

| Time | Owner | Entry Segment | Arrival Segment | |
|---|---|---|---|---|
| 1 | A | B+C | D | |
| 2 | B | C | A+D | |
| 3 | C | - | B+A+D | |
| 4 | B | A+D | C | |
| 5 | A | D | B+C | |
| 6 | D | - | A+B+C | |
| 7 | A | B+C | D | cycle repeats |

**Table 2: Example of palindromic admission schedule**

admission schedule, $A-B-C-D-D-C-B-A$ for instance, we will have fair admission over the long term, but threads $A$ and $D$ will enjoy lower LLC miss rates than $D$ and $C$, imposing a different form of unfairness related to residual cache residency. The palindrome schedule, however, enjoys better overall aggregate LLC miss and throughput than a simple repeating round-robin FIFO schedule of $A-B-C-D-A-B-C-D$. And in fact the FIFO schedule is pessimal, if we require equal fairness as measured over two back-to-back cycles [7].

Using the scenario above, and a simplistic decay mode, when a thread ceases waiting and takes ownership of the lock, it incurs a "cache reload transient"[45] where it suffers a burst of cache misses as it reprovisions the LLC with its own previouisly displaced private data. The residual residency fraction can be approximated as $Residual(T) = exp(-x * Lambda)$ where $T$ is the sojourn or waiting time – the number of quanta since the thread last ran – and *Lambda* parameterizes the decay rate (usually expressed as half-life). As *Residual* is a convex function, we can employ Jensen's inequality [5] as follows. Taking thread $B$ as a specific example, its waiting times under the FIFO schedule is always 3 time units and under the palindrome schedule the waiting time alternate 2-4-2-4 etc. The average waiting time is the same under both schedules, but the average residual LLC residency when the thread resumes is higher under the palindome schedule as $Residual(2) + Residual(4) \leq Residual(3) + Residual(3)$. Higher residency fractions implies reduced miss rates and better performance. We assume the critical section accesses both shared global as well as thread-private data.

Intuitively, an alterating sequence that has a short waiting time $P$ following by a longer waiting time $Q$ is better in terms of cache misses, than a sequence that uses a fixed waiting time of $(P+Q)/2$, Given exponential decay, the benefits accrued by the short delay $Q$ outweigh the decay penalty of the longer delay $P$ as most of the decay occurs early in the waiting phase.

## A.4 Potential Throughput Benefits – Aggregate Miss Rate

Crucially, under the palidrome schedule, threads can incur disparate cache hits rates, reflecting a form of long-term cache-based unfairness, even if they the admission is long-term fair. The overall aggregate miss rate for the palindrome schedule, however, as computed over all the threads, will be less than that in the round-robin FIFO schedule. We note in passing that a random admission order is statistically long-term fair for both admission frequency and cache residency, and also will display a lower aggregate cache miss rate than FIFO. Finally, the same effects that apply to cache residency also have analogs in page working sets.

## A.5 Mitigation

If desired, we can mitigate the unfairness from the effects above in a number of ways. A simple and expedient approach is to stochastically disrupt or perturb the repeating cycle, which reestablishes statistical long-term fairness. A viable technique is for incoming owners, having just acquired the lock, to run a thread-local Bernoulli trial, and based on the outcome, occasionally defer and immediately cede ownership to the next element in the entry segment, and propagate a reference to its wait element through the entry segment, where it will percolate to the tail, and eventually be re-granted ownership. This modification does not abrogate or otherwise violate our bypass guarantees or imperil anti-starvation as the reording is strictly intra-segment.

More generally, if we just pick random elements from the entry segment for succession, we still provide the same desirable population bounded anti-starvation property, statistically avoid long-term admission unfairness and cache residency fairness, but still enjoy short-term cache residency benefits.

[7] Yuan et al. [52] use the term "sawtooth" for the palindrome schedule

# B ON-STACK ALLOCATION OF WAIT ELEMENTS

As described above, our implementation places wait elements in thread-local storage. This simplifies reasoning about memory correctness as the tenure and lifespan of thread-local storage is the same as that as the associated thread. We observe, however, that in many environments, wait elements can also be safely allocated on-stack, in the activation frame of `Acquire`. This approach reduces memory usage to $K * W + L * B$ where $K$ is the number of *waiting* threads, $W$ is the size of the waiting element in the stack frame, $L$ is the number of currently extant locks, and $B$ is the size of the lock body.

At a high level, the element on which threads spin requires a short lifespan (tenure) and is only required to exist and remain in scope for the duration of the `Acquire` operation, and as such can be allocated in the `Acquire` function's frame. This is more convenient in terms of lifecycle than CLH or MCS.

But, in some circumstances – zombies – we also use the address of an element as an end-of-segment marker. In this case the address escapes the frame as it passed through the `Terminus` field. Specifically, the address of the wait element has a longer lifespan than the wait element itself, and the address persists even after the wait element has fallen out of scope. (Note that we use address-based comparisions to detect the end-of-segment, but never actually reference the defunct wait elements).

An address escaping its frame or scope, and having a longer tenure or lifespan that its referent, is considered "undefined behavior" in C++. In practice, though, we believe the technique is viable. If a given virtual address stack address remains associated with a given thread for the lifetime of the thread, as is the common case for pthreads environments, then the approach is safe. We note, however, that such on-stack allocation might not be safe under exotic non-standard thread models where multiple lightweight threads can run or be "mounted" on a given stack at different points in time, and stack addresses do not map to or otherwise convey thread identity in a stable fashion. If this particular aspect is a concern, the implementation could simply opt to adhere to our basic design and place the wait element into thread-local storage (TLS), which completely obviates the issue.

We also note that modern compilers can detect such behavior, and generate warnings, and in some cases, to help protect against undefined behavior, will actively annul references that can be statically determined to fall out of scope.

# C VARIATIONS

## C.1 Annotated Version

Listing-2 is an expanded and annotated version – with extra comments and `assert` invariants – of Listing-1, which is somewhat abridged.

```
1  struct Element {
2  |  std::atomic<int> Gate {0} ;
3  |  std::atomic<Element *> Terminus {nullptr} ;
4  } ;

5  // Encoding for Arrivals :
6  // 0:0 = Unlocked
7  // 0:1 = Locked with empty arrival list : SIMPLELOCKED
8  // T:0 = Locked with populated arrival list where T is the
9  //       most recently arrived thread on the arrival stack

10 struct Lock {
11 |  std::atomic <Element *> Arrivals {nullptr} ;
12 } ;

13 static auto const SIMPLELOCKED = (Element *) uintptr_t(1) ;

14 static Element * Acquire (Lock * L) {
15 |  static thread_local Element E {} ;
16 |  E.Terminus.store (nullptr, std::memory_order_release) ;
17 |  E.Gate.store   (0     , std::memory_order_release) ;

18 |  auto tail = L→Arrivals.exchange (&E) ;
19 |  assert (tail ≠ &E) ;
20 |  if (tail == nullptr) {
21 |  |  // fast-path uncontended acquire -- We now hold the lock
22 |  |  // First, try to undo or recover the above and replace &E
23 |  |  // with SIMPLELOCKED.
24 |  |  // The downside to this tactic is that we have 2 atomic SWAP
25 |  |  // operations in the fast uncontended path, putting more
26 |  |  // coherence traffic on the central arrival L→Arrivals word
27 |  |  // and increasing RMR (remote memory reference) complexity.
28 |  |  // But the two SWAP operations execute nearly back-to-back so
29 |  |  // we have very high odds that the underlying cache line
30 |  |  // remains in "modified" M-state.
31 |  |  // Our RMR complexity increases but in practice that is not
32 |  |  // necessarily of consequence for performance.
33 |  |  // Also, the double SWAP only occurs in the absense of
34 |  |  // contention. We care much more about coherence misses and
35 |  |  // write invalidations as a figure of merit.
36 |  |  // Critically, the arrival doorway remains wait-free and our
37 |  |  // desired bypass/overtaking properties remain unchanged.
38 |  |  // We hold the lock so anything we do after the 1st exchange
39 |  |  // effectively increases the lock hold time and may act
40 |  |  // to decrease ultimate scalability.
41 |  |  auto R = L→Arrivals.exchange (SIMPLELOCKED) ;
42 |  |  assert (R ≠ nullptr) ;
43 |  |  if (R == &E) return nullptr ;
44 |  |
45 |  |  // Other threads arrived and pushed onto L→Arrivals in
46 |  |  // the Exchange-Exchange window, above.
47 |  |  // Our &E is now burried in the stack.
48 |  |  // More precisely, our &E resides at the distal end of
49 |  |  // the arrival stack
50 |  |  // We expect this scenario to be relatively rare.
51 |  |  // We defer removal of &E and resort to the Terminus
52 |  |  // mechanism.
53 |  |  // Our 2nd exchange, above, decapitated and detached
54 |  |  // that list of recently-arrived-threads.
55 |  |  // &E resides at the end of that detached segment.
56 |  |  // We form a new detached entrylist with that segment
57 |  |  // We are and remain the lock owner so it is safe to
58 |  |  // detach.
59 |  |  //
60 |  |  // Note that a thread's element address can appear in both
61 |  |  // the arrival list and at the same time on the entrylist
62 |  |  // as the stale terminal element. This is benign.
63 |  |  assert (R→Gate.load() == 0) ;
64 |  |  assert (R→Terminus.load() == nullptr) ;
65 |  |  R→Terminus.store (&E, std::memory_order_release) ;
66 |  |
```

```
        // Following is not required but provides improved assert hygiene
        // No thread should subsequently access E
        E.Gate = 1 ;

        return R ;
    }

    // Coerce SIMPLELOCKED to nullptr
    // prev will be our successor when we subsequently unlock()
    auto prev = (Element *) (uintptr_t(tail) & ~1) ;
    assert (prev ≠ &E) ;

    // slow path : contention -- waiting phase
    while (E.Gate.load() == 0) {
        Pause() ;
    }

    // Determine if we are the end of chain and if
    // necessary propagate the eos end-of-chain marker value
    auto eos = E.Terminus.load() ;
    assert (eos ≠ &E) ;

    // The following "if" isn't strictly necessary as the case it
    // covers is also subsubsumed by the subsequent (EoS == prev)
    // test, but it allows better asserts
    // It also helps code comprehension and makes invariants
    // clearer
    if (prev == nullptr) {
        assert (eos == nullptr) ;
        return nullptr ;
    }

    if (prev == eos) {
        // We have reached the tail marker in the entrylist segment
        // Critically, this is an address-based check
        // Recognize the deferred removal of burried element
        // This also covers the case of normal null-terminated
        // end-of-segment
        return nullptr ;
    }

    // Propagate eos value thru the detached entrylist chain
    assert (prev ≠ nullptr) ;
    assert (prev→Gate == 0) ;
    assert (prev→Terminus.load() == nullptr) ;
    prev→Terminus.store (eos, std::memory_order_release) ;
    return prev ;
}

void Release (Lock * L, Element * succ) {
    assert (L→Arrivals.load() ≠ nullptr) ;

    // Case : entrylist populated -- appoint successor from entrylist
    // If a successor exists on implicit Entrylist,
    // just pass ownership to that waiting thread.
    if (succ ≠ nullptr) {
        assert (succ→Gate == 0) ;
        succ→Gate.store (1, std::memory_order_release) ;
        return ;
    }

    // Case : entrylist and arrivals are both empty
    if (L→Arrivals == SIMPLELOCKED) {
        // fast-path uncontended unlock
        auto v = L→Arrivals.cas (SIMPLELOCKED, nullptr) ;
        if (v == SIMPLELOCKED) return ;
        // The cas() failed :
        // The only possible transition on L→Arrivals is from
        // SIMPLELOCKED to populated, because of concurrent
        // arrivals in the LD-CAS window.
        // So we just fall through ...
        assert (v ≠ nullptr) ;
    }

    // Case : entrylist is empty and arrivals is populated
    // New threads have arrived and pushed themselves onto the
    // arrival stack.
    // We now detach that segment, shifting those arrivals to
    // become the next entrylist segment
    auto w = L→Arrivals.exchange (SIMPLELOCKED) ;
    assert (w ≠ nullptr) ;
    assert (w ≠ SIMPLELOCKED) ;
    assert (w→Terminus.load() == nullptr) ;
    assert (w→Gate.load() == 0) ;
    w→Gate.store (1, std::memory_order_release) ;
}
```

**Listing 2: Reciprocating Locks – Annotated**

## C.2 Simplified with ownership relay

Listing-3 reflects a somewhat simplified variant on Listing-2. This variation still employs the *double swap* technique, but avoids the use of the end-of-segment `terminus` marker, yielding a simplified algorithm. Instead, if additional threads raced and arrived and pushed onto the arrival segment in the window between the two exchange operations, the current owner simply abdicates and relays ownership to the first thread in the newly detached entry list, and then proceeds to wait. While simpler, we believe this variant has poorer progress properties, as, when the race manifests, ownership needs to relay through the race victim thread.

In this particular variation we can safely allocate the waiting elements on stack, if desired.

```
struct Element {
  std::atomic<int> Gate {0} ;
  std::atomic<Element *> Terminus {nullptr} ;
} ;

// Encoding for Arrivals :
// 0:0 = Unlocked
// 0:1 = Locked with empty arrival list : SIMPLELOCKED
// T:0 = Locked with populated arrival list where T is the
//       most recently arrived thread on the arrival stack

struct Lock {
  std::atomic <Element *> Arrivals {nullptr} ;
} ;

static auto const SIMPLELOCKED = (Element *) uintptr_t(2) ;

Element * Acquire (Lock * L) {
  Element E {} ;
  auto tail = L->Arrivals.exchange (&E) ;
  assert (tail ≠ &E) ;
  if (tail == nullptr) {
    // fast-path uncontended acquire
    auto R = L->Arrivals.exchange (SIMPLELOCKED) ;
    assert (R ≠ nullptr) ;
    if (R == &E) return nullptr ;

    // Other threads arrived and pushed onto L->Arrivals in
    // the Exchange-Exchange window, above
    // Our &E is now burried in the arrival stack.
    // We expect this scenario to be relatively rare.
    //
    // We recover and compensate by passing or diverting ownership
    // directly to the thread associated with R. Our own element,
    // E, which resides within the now-detached stack, will k
    // eventually gain ownership via natural succession.
    // R is the most-recently-arrivated thread (at the time of the
    // 2nd exchange, above) and is the head of that detached stack.
    //
    // This form arguably does _not provide a classic bounded
    // wait-free transfer of ownership, as, in this path, ownership
    // needs to pass transiently through this thread as an
    // intermediary before reaching R.
    // We relay ownership immediately to R.
    // Succession and progress depend on this thread making
    // progress to enable the actual successor, R.
    //
    // And while simpler, this approach can generate
    // more coherent communication traffic as ownership bounces
    // from this thread and then immediately to the intended
    // successor, R.
    assert (R->Gate == 0) ;
    R->Gate.store (1, std::memory_order_release) ;
    // fall through into normal waiting ...
  }

  // Coerce SIMPLELOCKED to nullptr
  auto prev = (Element *) (uintptr_t(tail) & ~1) ;
  assert (prev ≠ &E) ;

  // slow path : contention -- waiting phase
  while (E.Gate.load() == 0) {
    Pause() ;
  }

  return prev ;
}

void Acquire (Lock * L, Element * succ) {
  assert (L->Arrivals.load() ≠ nullptr) ;

  // If successor exists on implicit Entrylist,
  // just pass ownership to that waiting thread.
  if (succ ≠ nullptr) {
    assert (succ->Gate == 0) ;
    succ->Gate.store (1, std::memory_order_release) ;
    return ;
  }

  if (L->Arrivals == SIMPLELOCKED) {
    // fast-path uncontended unlock
    auto v = L->Arrivals.cas (SIMPLELOCKED, nullptr) ;
    if (v == SIMPLELOCKED) return ;
  }

  // Decapitate the arrival segment, leaving the lock held.
  // Reprovision the entrylist with those arrivals.
  auto w = L->Arrivals.exchange (SIMPLELOCKED) ;
  assert (w ≠ nullptr) ;
  assert (w ≠ SIMPLELOCKED) ;
  assert (w->Gate.load() == 0) ;
  w->Gate.store (1, std::memory_order_release) ;
}
```

**Listing 3: Reciprocating Locks – simplified**

## C.3 Double-swap Avoidance

Listing-4 illustrates a form that avoids the need for the *double swap* on arrival, albeit at the expense of more ornate encodings of the value passed from `Acquire` to `Release`, which can now reflect either the successor, or if a successor is not present, the end-of-segment sentinel address. The end-of-segment address markers now need to flow through the `Release` operation so the marker can be extricated in a deferred fashion.

```
struct Element {
  std::atomic<int> Gate {0} ;
  std::atomic<Element *> Terminus {nullptr} ;
} ;

struct Lock {
  std::atomic <Element *> Arrivals {nullptr} ;
} ;

// Encoding for Arrivals :
// 0:0 = Unlocked
// 1:0 = Locked with empty arrival list : SIMPLELOCKED
// T:0 = Locked with populated arrival list where T is the
//       most recently arrived thread in the arrival stack

static auto const SIMPLELOCKED = (Element *) uintptr_t(2) ;

Element * Acquire (Lock * L) {
  static thread_local Element E {} ;
  E.Terminus.store (nullptr, std::memory_order_release) ;
  E.Gate.store (0, std::memory_order_release) ;

  auto tail = L->Arrivals.exchange (&E) ;
  assert (tail ≠ &E) ;
  if (tail == nullptr) {
    // fast-path uncontended acquire
    // E should never be subsequently accessed as immediately falls
    // out of scope when control returns from Acquire().
    // The address of E, which acts as an end-of-segment "EoS"
    // marker, has a longer tenure than E itself, but &E can not
    // recycle within the window where it could reappear on a
    // segment, as the address is tied to the calling thread.
    // Invariant : E will never re-appear in the same segment.
    // The only exception would be the case where a thread dies
    // holding a lock acquired via the fast path and then dies,
    // allowing &E to recycle in the context of a new thread,
    // and the new thread tries to acquire with E.
    // That's deadlock anyway.

    // Following is not required but provides improved assert hygiene
    E.Gate = 1 ;

    // Convey terminus marker value to corresponding unlock()
    // &E is a temporally unique and thread-specific "nonce"
    // identifier.
    return &E ;
  }

  // Coerce SIMPLELOCKED to nullptr
  // prev will be our successor when we unlock()
  auto prev = (Element *) (uintptr_t(tail) & ~3) ;
  assert (prev ≠ &E) ;

  // slow path : contention -- waiting phase
  while (E.Gate == 0) {
    Pause() ;
  }

  // Determine if we are the end of chain and if
  // necessary propagate the eos end-of-chain marker value
  auto eos = E.Terminus.load() ;
  assert (eos ≠ nullptr) ;
  assert (eos ≠ &E) ;

  // The following "if" isn't strictly necessary as the case it
  // covers is also subsubsumed by the subsequent (eos == tail)
  // test, but it allows better asserts.
  // It also helps code comprehension and makes invariants clearer
  if (prev == nullptr) {
    assert (eos == SIMPLELOCKED) ;
    return SIMPLELOCKED ;
  }

  if (eos == tail) {
    // We have reached the tail marker in the entrylist segment
    return SIMPLELOCKED ;
  }

  // Propagate eos value thru the detached entrylist chain
  assert (prev ≠ nullptr) ;
  assert (prev->Terminus == nullptr) ;
  prev->Terminus = eos ;
  return (Element *) (uintptr_t(prev) + 1) ;
}

// Encoding for "s" which is passed from Lock() to unlock()
// E:0 and 1:0 intentionally align with and "rhyme" with
// L->Arrivals encoding.
// Invariant : s must be non-zero !
// S:1 = EntryList is populated -- S is successor in EntryList
// E:0 = EntryList is empty  -- specific element E is end-of-segment
// 1:0 = EntryList is empty  -- Normal SIMPLELOCKED
// 0:0 = Illegal and undefined
// 0:1 = Illegal and undefined

void Release (Lock * L, Element * s) {
  assert (L->Arrivals.load() ≠ nullptr) ;

  // s is either an EoS marker or a successor,
  // distinguished by the least significant bit of "s"
  assert (s ≠ nullptr) ;

  // If successor exists on implicit entrylist,
  // just pass ownership to that waiting thread.
  if (uintptr_t(s) & 1) {
    s = (Element *) (uintptr_t(s)-1) ;
    assert (s ≠ nullptr) ;
    assert (s->Terminus ≠ nullptr) ;
    assert (s->Gate == 0) ;
    s->Gate.store (1, std::memory_order_release) ;
    return ;
  }

  // s is the EoS marker
  // EoS will either be SIMPLELOCKED or the address of a stale
  // element that was posted onto L->Arrival on the last
  // uncontended fast-path arrival
  auto eos = s ;
  if (L->Arrivals.load() == eos) {
    // fast-path uncontended unlock
    auto v = L->Arrivals.cas (eos, nullptr) ;
    if (v == eos) return ;
    // The cas() failed :
    // The only possible transition on L->Arrivals is from eos to
    // populated, because of concurrent arrivals in the LD-CAS
    // window.
    // So we just fall through ...
  }

  // New threads have arrived and pushed themselves onto the
  // arrival stack.
  // We now detach that segment, shifting the arrivals to
  // become the next entrylist segment
  //
  // If we originally acquired the lock via the fast-path but
  // subsequent threads arrived and pushed, then our original
  // element is covered or "burried" on the arrival stack and
  // is the final element.
  // We just move arrival segment to the entrylist and skip
  // that element when we are about to pass ownership to it, by
  // recognizing the address.
  // Critically, we never access the element.
  // Note that a thread's element address can appear in both the
  // arrival list and at the same time on the entrylist as the
  // stale terminal element.
  auto w = L->Arrivals.exchange (SIMPLELOCKED) ;
  assert (w ≠ nullptr) ;
```

```
    assert (w ≠ SIMPLELOCKED) ;
    assert (w ≠ eos) ;
    assert (w→Terminus.load() == nullptr) ;
    assert (w→Gate.load() == 0) ;
    w→Terminus.store (eos, std::memory_order_release) ;
    w→Gate.store (1, std::memory_order_release) ;
}
```

**Listing 4: Reciprocating Locks – double-swap avoiding**

## C.4 Retrograde Ticket Lock

Finally, in Listing-5 we describe the *retrograde ticket lock* algorithm, which provides the same admission order policy as Reciprocating Locks but implemented as a version of the classic ticket lock algorithm.

```
struct TicketLock {
    std::atomic <int> Ticket {0} ;
    std::atomic <int> Grant {0} ;
} ;

// Classic Ticket lock ...

void TicketAcquire (TicketLock * L) {
    auto ticket = L→Ticket.fetch_add(1) ;
    while (L→Grant ≠ ticket) {
        Pause() ;
    }
}

void TicketRelease (TicketLock * L) {
    auto grant = L→Grant.load() + 1 ;
    L→Grant.store (grant, std::memory_order_release) ;
}

// "Retrograde" ticket lock
// provides the same admission order as reciprocating locks
// Note that the Acquire() operator is identical to that of
// the baseline ticket lock, above.
// All changes relative to ticket locks are encapsulated
// in the unlock operator.
//
// Invariant : Ticket >= Top >= Grant >= Base
// Threads with tickets values in [Top,Base] are admitted in reverse order
// Waiting  : Ticket thru Top union with Grant thru Base
// Completed : Top thru Grant union with those < Base
// EntrySet : Grant thru Base
// Arrivals : Ticket thru Top


struct RetrogradeLock {
    std::atomic <int64_t> Ticket {0} ;
    std::atomic <int64_t> Grant {0} ;
    std::atomic <int64_t> Top {0} ;
    std::atomic <int64_t> Base {0} ;
} ;

void RetrogradeAcquire (RetrogradeLock * L) {
    auto ticket = L→Ticket.fetch_add(1) ;
    while (L→Grant ≠ ticket) {
        Pause() ;
    }
}

void RetrogradeRelease (RetrogradeLock * L) {
    auto grant = L→Grant.load() - 1
    if (grant > L→Base) {
        // Region of reverse admission order ...
        // working our way backward through entry set
        L→Grant = grant ;
    } else {
        auto IncomingHi = L→Top.load() ;
        L→Base = IncomingHi ;
        auto tmp = L→Request.load() ;
        L→Top = tmp - 1 ;
        if (tmp == (IncomingHi + 1)) {
            // Apparently no waiters
            // Uncontended release -- set to "unlocked" state with
            // Ticket == Grant.
            // Benign if Ticket advances concurrently after we read it.
            // The store into L→Grant must become visible after the
            // other stores.
            L→Top  = tmp ;
            L→Base = tmp ;
            L→Grant = tmp ;
        } else {
```

```
63 |  |  |  // Contention with waiters -- stays locked
64 |  |  |  L→Grant = tmp - 1 ;
65 |  |  }
66 |  }
67 }
```

**Listing 5: Retrograde Ticket Locks**