Fast and Fair Lock-Free Locks

Naama Ben-David and Guy Blelloch

August 11, 2021

Abstract

We present a randomized approach for lock-free locks with strong bounds on time and fairness in a context in which any process can be arbitrarily delayed. Our approach supports a tryLock operation that is given a set of locks, and code to run when all the locks are acquired. Given an upper bound \( \kappa \) known to the algorithm on the point contention for a tryLock it will succeed in acquiring its locks and running the code with probability at least \( 1/\kappa \). It is thus fair. If the algorithm does not know the bound \( \kappa \), we present a variant that can guarantee a probability of at least \( 1/\kappa \log \kappa \) of success. Furthermore, if the maximum step complexity for the code in any lock is \( T \), and the point contentsions are constant, the attempt will take \( O(T) \) steps. The attempts are independent, thus if the tryLock is repeatedly retried on failure, it will succeed in \( O(T) \) expected steps, and with high probability in not much more. Importantly, however, retrying is not mandatory, and a process may choose to execute different code upon failure.

We assume an oblivious adversarial scheduler, which does not make decisions based on the operations, but can predetermine any schedule for the processes, which is unknown to our algorithm. Furthermore, to account for applications that change their future requests based on the results of previous lock attempts, we strengthen the adversary by allowing decisions of the start times and lock sets of tryLock attempts to be made adaptively, given the history of the execution so far.
1 Introduction

Mutual exclusion—i.e., executing a “critical region” of code so it appears to happen in isolation—is likely the most important problem in concurrent and distributed computing. It comes in many forms (e.g., locks, transactions, universal constructions). Here we are interested in the setting in which the user can simultaneously take multiple locks. Allowing for multiple locks is crucial in making a practical system; while having a single global lock that everyone must compete on is enough to ensure safety in any system, having fine-grained locks that protect small parts of the system allows non-conflicting accesses to proceed in parallel, asymptotically increasing throughput. A classic example is the dining philosophers problem in which a set of philosophers sit around a table with locks (often called forks) between them. Each philosopher needs to simultaneously acquire both of its neighboring locks to eat. In this way 1/2 the philosophers (rounded down) can eat at a time instead of just one of them if they used a single global lock. Perhaps more practically, one may want to lock a vertex and its neighbors in a tree or graph to make a local change on the neighborhood, or implement transactions that read and write some number of shared locations, each associated with a lock.

Although locks can be used to for mutual exclusion, locks have well-known problems both in theory and in practice. These problems stem from the inherent asynchrony in real systems; processes may execute at vastly different speeds, and may stall indefinitely between any two instructions. Thus, even if a process only executes a constant number of steps while holding a lock, it can greatly delay others in the system. To avoid this problem, researchers have developed a wide range of lock-free and wait-free algorithms that ensure progress even when processes stall and fail. However, developing such algorithms can be significantly more difficult than a lock-based algorithm.

To help solve this issue, Turek et al. [38], and independently Barnes [9], introduced the notion of lock-free locks. The basic idea is to allow processes to take locks, but to leave enough information behind to allow other processes to help complete the code inside the lock, so that the lock can be released. The approaches support taking multiple locks. To allow processes to help each other in a safe way, the code they run must be idempotent, meaning that even if it is run several times, it only takes effect once. This general approach has been used in several concurrent algorithms to convert lock-based algorithms to lock-free algorithms (e.g., [13, 19]), and to implement transactions in a lock-free manner [20,36]. However, these approaches still do not guarantee fairness—in particular, a process could repeatedly help another process complete, and then be beat to the next acquire by another process. To ensure fairness, a variety of wait-free approaches have been developed, but most of these effectively require sharing a single lock, fully sequentializing the execution [25]—e.g. the dining philosophers would all have their fair turn at eating, but would eat one at a time.

In this paper, we present a randomized approach for wait-free locks with strong bounds on time and fairness. Indeed they are the first bounds that show for a problem where all operations take a constant number of locks, and each lock has constant contention, that all locked operations will complete in a constant expected number of steps—i.e. all philosophers will eat in an expected constant number of their steps. After completing each proces, or philosopher, can repeat, eating multiple times waiting expected constant time each time. The randomization is used to break the symmetry in the problem and is based on using random priorities. The approach necessarily uses helping but avoids recursive helping—i.e., it only helps other processes that directly share a lock.

In our approach, a tryLock attempts to acquire some number of locks and, if successful, runs its critical code. We describe an algorithm that if a tryLock has bounded point contention $\kappa$, defined below, it will succeed with probability at least $1/\kappa$. This means that all contenders “fairly” compete on the locks—e.g., for the philosophers the bounded point contention is four (two locks each with contention two), so each philosopher’s attempt will succeed with probability at least ...
1/4. Furthermore if each tryLock has constant contention and its critical code runs in at most $T$ steps, then each attempt (whether successful or not) will take $O(T)$ steps (total instructions of the process running it including both memory and local instructions). The attempts by a process are probabilistically independent, so if a process keeps trying on a lock, with probability $p$ the lock will successfully complete in $O(T \log(1/p))$ total steps. This implies $O(T)$ expected time, and is sometimes referred to as randomized wait-freedom [14] since the probability of taking a long time is vanishingly small. Furthermore, we ensure that an attempt can only fail if another succeeds, so the approach is always lock-free. More precise bounds when contention varies across locks is given in the body of the paper.

In concurrent settings, an adversarial scheduler is often used to model the inherent asynchrony in concurrent systems; that is, the order in which processes execute steps (the schedule) is assumed to be controlled by an adversary. Adversarial scheduler models differ in how much information the adversary has about the execution. An adaptive adversary is assumed to see everything that has happened in the execution thus far, whereas an oblivious adversary is assumed to make all of its scheduling decisions before the execution begins. Some separations are known between adaptive and oblivious scheduler settings [1, 21, 22, 32]. Often, an oblivious adversary is considered to be a reasonable assumption, since asynchrony in real hardware is not generally affected by the values written on memory. When considering locks, in addition to the adversarial scheduler, the processes acquiring the locks can be adversarial, possibly trying to increase or decrease their probability of succeeding, or their number of steps. We refer to the processes that are invoking tryLocks as players. Once a player requests a lock, the steps taken might no longer be its control, but it can still decide when to make the request. An oblivious player adversary, is one that has pre-decided when each of its tryLock attempts will start. An oblivious player is not realistic in general since deciding on whether we take a lock almost always depends on previous outcomes. This is especially true for tryLocks since whether we decide to try a second time will often depend on previous attempts. Therefore, in this work we consider two separate adversaries; the player adversary, which is adaptive and controls the start time of each tryLock request, and the scheduler adversary, which is oblivious and controls the order of steps of processes with active tryLock requests.

When considering contention for a lock there are a few ways to measure it. Using relatively standard convention [8], the point contention for a lock at a given step is the number of tryLocks involving the lock that overlap the step. The point contention of a tryLock is the maximum across its steps of the sum of the point contention across its locks on that step. We note that this differs significantly from the interval contention, which for a tryLock $t$ is the total number of other tryLocks that overlap $t$ in time and share a lock. For example with the dining philosophers problem the point contention for each philosopher is at most four since it can have at most contention two on each of its two locks. However the interval contention is unbounded since while trying to grab a fork a neighboring agile philosopher could try grabbing that fork an unbounded number of times (the relative speeds of philosophers has no bound, and in fact one could fall asleep or worse). In a randomized setting the point contention can depend on the random choices making it hard to claim probability bounds for any actual point contention of a history. For example the adaptive player could keep throwing more contention at a lock until it loses. Therefore in this paper we consider bounded point contention, which for a lock is a bound the maximum point contention possible across all steps of the history, considering all random choices and all oblivious schedulers. For the philosophers this is still two, and for a static graph that locks a vertex and its neighbors it is still the degree plus one.

In this paper we first describe an algorithm that needs to know the bounded point contention for each lock. We then outline how to extend this to allow for unknown bounds. This requires some loss in the probability of tryLocks succeeding. In particular, for a tryLock with bounded contention
\( \kappa \) that is unknown to the algorithm the probability of success is \( \Omega\left( \frac{1}{\kappa \log \kappa} \right) \).

1.1 Overview and Challenges

Here we give an overview of the approach and motivate the challenges through some examples.

To abstract away from low-level details of our algorithm, we consider acquiring locks as a tournament. We later map this onto concrete machine instructions ensuring the invariants are maintained. Players (tryLocks) enter the tournament with some skill level (priority), requesting to participate in some set of events (locks). They compete in the events against other players, winning an event if they have a higher skill level. If they win all their events they celebrate (execute their critical region), and then leave the tournament. Once a player wins another player cannot win until the first player celebrates. While in the tournament the players take steps. The scheduler adversary chooses in what order to interleave the player steps—semantically they happen one at a time. The player adversary chooses when to have players enter the tournament, but once a player enters it follows a fixed protocol. In our terminology each player only plays once—if a thread retries to win the tournament (execute its critical region) it uses a new player (new tryLock).

Importantly we would like to ensure some form of fairness and bound the number of player steps needed to complete. Bounding the steps requires having players help other players. Without helping, in particular, a player could win and then the scheduler could not execute any more of its steps blocking other players from ever completing. To ensure fairness we would like to make the competition unbiased such that the other players a player potentially competes with (i.e. those in its point contention) are all equally likely to have maximum skill. Given this, its probability of winning would be at least inversely proportional to its point contention.

A difficulty with making the tournament unbiased involves leaks of information about active players, often due to timing “attacks”. To understand the difficulty, we consider some of the attacks that can take place in such a tournament. In the simplest case, and with just a single event, consider the case that a player takes longer to win than to lose. This will mean incoming players are not meeting an unbiased competitor—the competitor is more likely to be winning and hence have a higher priority. Thus, we need to ensure that winning and losing, or, more generally, the skill level of a player, does not affect that player’s time in the tournament.

With multiple events (locks) the problem is exasperated since there are many side channels to pass information. Here we give one example, although we could describe a dozen more. Consider player A competing in events \( x \) and \( y \), and player B competing in just \( x \). Player A has to compete one event at a time, and could first beat B on event \( x \) before competing on \( y \). This means information about A will be revealed to the adversary before A competes on \( y \), letting other adversarial players choose whether to compete against A based on how strong it is. Even if the adversary does not know information about players’ skill level until after they finish, B’s defeat could leak information about A. To avoid this, we could force B to help finish A (including its competition on \( y \)) before it finishes but this causes another problem. The issue is that this could cause recursive helping; maybe A would need to help another player C on \( y \), which in turn would cause B to help C. Such recursive helping chains could reach everyone in the tournament even though each player has low contention. Without forcing full synchronization (and hence unbounded helping) there are many ways that information can be leaked about active players, and trying to avoid this seems futile.

On the positive side, if skills (priorities) are selected randomly and if the set of players one competes with does not depend on skill, then the tournament will be fair (a player wins with probability inversely proportional to the number of competitors). Fortunately there are only a couple ways to change who a player competes against based on skill. One way is to change the number of steps taken by a player based on its skill, for example taking longer when the skill is
high. Such changes could be made by forcing the player to take extra steps (e.g. by having the player adversary introduce another competitor), or simply by a bad tournament protocol that takes more steps when winning than when losing. This can be fixed by ensuring the number of steps is not biased by the priorities. Another way to change the competition based on skill is to control when a player enters a tournament and competes in an event—e.g. making it enter when there is a skilled player in the tournament, and delaying it when players are weak. This is more challenging to fix.

Our approach is to completely erase information leaks about players’ skill levels by forcing a player to ‘throw out’ useful information once it starts. Note that since we assume an adaptive adversary, all of the scenarios described above (and worse), can happen. An adaptive adversary knows the exact skill level of each player the moment that skill level is known to the player itself. To protect against this, we ensure that after a player enters the tournament, and is forced to follow the protocol, all useful information is “erased”, and no more usable information is gained before actually competing. This is done in two steps. Firstly, when entering we require a player to help finish all the events it will compete on, clearing any possible highly skilled players it might know about. That is, any player \( p' \) whose skill level was known to the adversary before player \( p \) joined the tournament will be forced to finish competing without competing against \( p \). Furthermore, \( p \) doesn’t pick its skill level (and therefore its skill level is not known to the adversary), until after it finishes this helping phase. This is not sufficient, however, since new players can come in after \( p \) and, if we are not careful, their skill level can possibly affect the timing of when \( p \) competes, dragging them into an event at inopportune times. Therefore, we force the player to wait for a fixed number of its own steps from its start (enough steps to ensure it can complete all of its helping), and then atomically makes itself visible to other players in the tournament—this is the point when it reveals its skill level and starts competing and being helped by others.

The important aspect of introducing these fixed delays is that the “reveal” time is unaffected by other players. It therefore cannot be sucked earlier or pushed later based on the skill level of competitors. We note that this approach is robust against strong adversarial players, but only an oblivious scheduler. A strong scheduler could still move the point at which a player becomes visible based on known skill levels.

2 Related Work

Using randomization to acquire locks is a hard problem that has been studied for many years. The difficulty arises from the lack of synchronization among processes, and the ability to delay processes based on observations of the current competition. Rabin [32] first considered the problem for a single lock, and only for the acquisition (i.e. no helping). Like ours, his scheme used priorities. Saias [33] showed the algorithm did not satisfy the claimed fairness bounds due to information leaks of the sort described in Section 1.1. Kushilevitz and Rabin [29] fixed it with a more involved algorithm. Lehmann and Rabin also developed an algorithm for the dining philosophers problem [30]. Lynch, Saias and Segala [31] later proved that with probability 1/16 within 9 rounds one philosopher would eat. However, our goal is much stronger, requiring a constant fraction to eat. Also the model was not fully asynchronous—a round involved every process taking a step. Duflot, Fribourg, and Picaronny generalized the algorithm to the fully asynchronous setting [18], but at the cost of a bound that depended on the number of processors. More recent work has also looked at randomized mutual exclusion for a single lock and without helping [21, 22].

It has been shown that a tenet of concurrent algorithm design, linearizability [26] does not nicely extend when randomization is introduced [23]. Linearizability allows operations that take
multiple steps to be treated as if they run atomically in one step. Unfortunately, however, analyzing probability distributions for the single step case does not generalize to a multistep linearized implementation, especially when analyzed for a weaker adversary. This is what lead to some of the difficulties encountered by the previous work, and some of the challenges we face.

Turek et al. [38] and independently Barnes [9] introduced the idea of lockless locks. They are both based on the idea of leaving a pointer to code to execute inside the locks, such that others can help complete it. In the locked code, Turek et al.’s method supports reads and writes and nested locks and unlocks. As with standard locks, cycles in the inclusion graph must be avoided to prevent deadlocks. Their approach thus allows arbitrary static transactions via two-phase locking by ordering the locks, and acquiring them in that order. It uses recursive (or “altruistic”) helping in that it recursively helps transactions encountered on a required lock. It is lock free, uses CAS, and although the authors do not give time bounds it appears that if all transactions take at most \( T_m \) time in isolation, the amortized time per transaction is \( O(pT_m) \). Barnes’s approach supports arbitrary dynamic transactions in a lock free manner, and uses LL/SC. It uses a form of optimistic concurrency [28] allowing for dynamic transactions. As with the Turek et al. approach, it uses recursive helping. Neither approach is wait-free—a transaction can continuously help and then lose to yet another transaction.

Shavit and Touitou [36] introduce the idea of “selfish” helping. They argue that if a transaction encounters a lock that it is taken, it should help the occupant get off the lock, but not recursively help. In particular, if while helping another transaction, it encounters a taken lock, then it aborts the transaction being helped. The approach only supports static transactions since it needs to take locks in a fixed order. It differs from our helping scheme in that there are no priorities involved. In our scheme when a transaction being helped meets another transaction on a lock, we abort the one with lower priority and continue with the one being helped if it is not the one aborted. Shavit and Touitou’s approach is again lock-free but not wait-free. The worst case time bounds are significantly weaker than Turek et al. or Barnes since there can be a chain of aborted transactions as long as the size of memory, where only the last one succeeds.

Fraser and Harris [20] extend Barnes’s approach based on optimistic concurrency and recursive helping. The primary difference is that they avoid locks for read-only locations by using a validate phase (as originally suggested by Kung and Robinson [28]). They break cycles between a validating read and a write lock on the same location, by giving arbitrary (not random) priorities to the transactions to break this cycle. As with Shavit and Touitou’s method, operations can take amortized time proportional to the size of memory. There has been a variety of work on contention management for transactions under controlled schedulers, some of it using randomization [7,24,34,35], but it does not apply to the asynchronous setting we are considering.

Starting with Herlihy [25], many researchers have studied wait-free universal constructions, many of which can be applied to at least a single lock, but most of these have an \( O(P) \) factor in their time complexity, where \( P \) is the total number of processes in the system, meaning that even under low contention they are very costly. Afek et al. [2] describe an elegant solution for a universal construction, or single lock in our terminology, that reduced the time complexity to be proportional to the point contention instead of the number of processors. Attiya and Dagan [6] describe a technique that should be able to support nested locks, although described in terms of operations on multiple locations. They only support accessing two locations (i.e., two locks). Considering the conflict graph among live transactions, they describe an algorithm such that when transactions are separated by at least \( O(\log^* n) \) in the graph, they cannot affect each other. The approach is lock free, but not wait free, and no time bounds are given. Afek et al. [3] generalize the approach to a constant \( k \) locks (locations) and describe a wait-free variant using a Universal construction. They show that the step complexity (only counting memory operations) is bounded.
by a function of the contention within a neighborhood of radius $O(\log^* n)$ in the conflict graph. Both approaches are very complicated due to their use of a derandomization technique for breaking symmetries [15].

3 Model and Preliminaries

In this paper we use standard operations on memory including Read, Write, and CAS. Beyond memory operations, processes do local operations (e.g. register operations, jumps, ...). Whenever we discuss the execution time for a process, we mean all operations (i.e., instructions) that run on the process including the local ones.

A procedure is a sequential procedure with an invocation point (possibly with arguments), and a response (possibly with return values). A step is either a memory operation or an invocation or response of a procedure. We assume all steps are annotated with their arguments and return values, and we say two steps are equivalent if these are the same. We say a memory operation has no effect if it does not change the memory (e.g. a read, a failed CAS or a write of the same value). We assume the standard definition of linearizability [26].

The history of a procedure is the sequence of steps it took, or has taken so far. The history of a concurrent program is some interleaving of the histories of the individual procedures. A history is valid if it is consistent with the semantics of the memory operations. A history differs from a schedule in that a schedule does not specify the random values processes get, but a history does. Thus, a schedule maps to several different histories in randomized protocols. There is a one-to-one mapping between histories and schedules in deterministic protocols.

In the paper we are interested in point contention for each tryLock attempt. We say a tryLock attempt is live on a lock $l$ from its invocation to its response (inclusive). The point contention for a tryLock attempt is the maximum number of live attempts on all of the locks in its lock set at any single point over its steps from its invocation to its response. For randomized protocols, the point contention of a tryLock attempt $p$ in a schedule $S$ is the maximum point contention that is possible for $p$ in $S$ across all realizations of the randomness (across all histories that $S$ can produce). When we place bounds on point contention, we mean that for no valid history of a program, can the lock have a higher point contention.

A thunk is a procedure with no arguments [27]. Note that any procedure along with given arguments can be converted into a thunk by wrapping the procedure with its arguments into a closure [37] (e.g. using a lambda in most modern programming languages). Here, for simplicity, we also assume thunks do not return any values—they can instead write a result into a specified location in memory. A thunk runs with some local private memory, and accesses the main memory via a fixed set of memory operations.

We assume two adversaries; an adaptive player adversary and an oblivious scheduler adversary. Formally, the scheduler adversary is a function from a time step to the process that runs an instruction on that time step, which produces a schedule. Our algorithms do not know what this function is, and the scheduler can delay any given process for an arbitrary length of time. The player adversary is a function from the history of an execution and a given process to a boolean indicating whether the given process starts a new TryLock at its next step.

3.1 Idempotence

To allow processes to help each other complete their thunks on a lock, we must ensure that regardless of how many processes execute a thunk, it only appears to execute once. For this, we use the notion of idempotence, which roughly means that a piece of code that is applied multiple times appears
as if was run once \[10,12,15,17\]. We define the notion of idempotence here, and show how to make any thunk involving Read, Write and compare-and-modify (CAM) instructions into one that is idempotent with constant overhead in Appendix A. A CAM is the same as a CAS, but does not return any value. Ben-David et al \[10\] show how to convert any CAS into a CAM using only reads, writes, and CAS operations. We therefore use CAM in our discussion of idempotence, but note that this can be implemented using the standard primitives we assume for this paper.

A thunk \(T\) generates a sequence of steps \(S_t\), which can depend on the memory state it sees. A step \(s \in S_t\) is said to be a step for thunk \(T\), regardless of which process executes it. A run of a thunk \(T\) is the sequence of steps taken by a single process to execute or help execute \(T\). The runs for a thunk can be interleaved. An instantiation of a thunk \(T\) is a subsequence of the steps for thunk \(T\) in a history \(H\), possibly from many different runs, such that those steps are consistent with a single run of the thunk. By consistent we mean all operations on those steps have the same arguments and return values.

**Definition 1** (Idempotence). A thunk \(T\) is idempotent if in any valid history \(H\),

1. there exists a valid instantiation \(H'\) of \(T\) that is a (possibly empty) subsequence of all operations from runs of \(T\) in \(H\),
2. if there is a finished run of \(T\) (response on \(T\)), then the last step of the first such finished run must be the end of \(H'\),
3. all memory operations for \(T\) in any of its runs in \(H\) that are not in \(H'\) have no effect on the shared memory.

The definition essentially states that the combination of all runs of a thunk \(T\) is equivalent to having run \(T\) once, and finishing at the response of the first run.

In Appendix A, we describe a simulation/translation that converts thunks involving Read, Write and compare-and-modify (CAM) instructions into one that is idempotent, proving the following result.

**Theorem 3.1.** Any thunk using constant sized local memory and using only Read, Write and CAM operations on shared memory can be simulated using Reads, Writes and CAMs as primitive operations such that (1) it is idempotent, and (2) every simulated memory operation takes constant time. As long as there are no concurrent writes and CAMs on the same location, the simulation is linearizable.

### 3.2 Mutual Exclusion with Idempotence

When implementing a tryLock, the safety property we require is as follows.

**Definition 2** (Mutual Exclusion with Idempotence). If a tryLock attempt with thunk \(T\) and lock set \(\mathcal{L}\) succeeds, then \(T\) was executed to completion at least once. Furthermore, the first time \(T\) was executed to completion was not concurrent with the execution of any other thunk whose lock set overlaps \(\mathcal{L}\).

### 4 An Adaptive Active Set Algorithm

We now present an algorithm for an active set object. This object will be useful in implementing our locking scheme; in Section 5, we show how to implement fast and fair locks by representing them as active set objects.
The active set object was first introduced in FOCS’99 by Afek et al. [3]. It has three operations; insert, remove, and getSet. A getSet operation returns the set of elements that have been inserted but not yet removed. Insert and remove operations simply return ‘ack’. The original definition was meant to keep track of membership; the insert and remove operations were called join and leave respectively, and these operations did not take any arguments. We generalize the problem slightly to allow the insert and remove operations to take an argument specifying an element to be inserted into the set, and specifying which element should be removed. Our version of the active set object is used in a similar way to the original; a process must alternate calling insert and remove, starting with insert, and must always remove the element that it most recently inserted.

We present pseudocode implementing the active set object in Algorithm 1. An announcements array of $C$ slots is maintained, where $C$ is the maximum number of elements that can be in the set at any given time. Each slot has an owner element and a set, which is a pointer to a list of elements. To insert an element, a process traverses the announcements array from the beginning, looking for a slot whose owner field is empty. It then takes ownership of this slot by CASing in its new element into the slot’s owner field. To remove an element from slot $i$, a process simply changes the owner of slot $i$ to null. We assume that a process maintains the index of the slot that it successfully owned in its last insert operation, and uses this index in its next call to remove. Intuitively, the owner fields of all the slots make up the current active set. An insert operation will always be able to find a slot without an owner, since there are $C$ slots, where $C$ is an upper bound on the number of elements in the active set at any time.

To help implement an efficient linearizable getSet function, the insert and remove operations propagate the changed ownership of their slot to the top of the announcements array by calling the climb helper function. The climb function works as follows. Starting at the slot given as an argument, it traverses the list to the top, replacing the set field of the current slot $i$ with the set of the previous slot $i + 1$, plus the owner of slot $i$. That is, the climb function intuitively collects all owners of the slots and propagates all of them to the set field of slot 0. The getSet function can then simply read the set of announcements[0] to get the current active set.

This algorithm is similar to the universal construction presented by Afek et al. in STOC’95 [2]. However, while their algorithm propagates operations to the top of an array to get executed, ours propagates elements to be added to the active set. Similarly to their algorithm, our active set algorithm is adaptive; the step complexity of the insert and remove operations is proportional to the size of the active set plus the point contention during the insert operation in a given execution. This is because the number of slots that an insert operation traverses before finding one with no owner is at most the number of elements currently in the active set, plus the ones in the process of being inserted. We note that when using the active set object to count membership in a larger context (as was its original intent and is the way we use it for the lock algorithm), this translates to the point contention in the larger context.

**Correctness and Step Complexity.** We show that the active set algorithm presented in Algorithm [1] is linearizable and insert/remove operations take at most a number of steps proportional to the size of the active set plus the point contention, which getSet takes constant steps. The correctness and step complexity arguments for Algorithm [1] are similar to the ones for Afek et al.’s universal construction [2]. However, full proofs of that algorithm have never been published, so for completeness we provide the full proof of our algorithm.

Intuitively, the climb(i) function ensures that any owner of a slot $0 \leq j \leq i$ makes it to slot 0’s set by the end of the climb(i) call. The time at which the owner of slot $i$ is added to slot 0’s set is the insert and remove operation’s linearization point. There are subtleties in arguing about
Algorithm 1: Active Set Algorithm

which owner of a slot $j$ makes it to the top of the array, since owners of a slot can change over time with concurrent insertions and removals.

Furthermore, we show that if an insert operation executed is successful CAS on the owner of slot $i$, then there were at least $i + 1$ elements in the active set or currently being inserted at some point during that operation. We show that the `climb` function can be implemented in a constant number of steps per iteration, by appending the owner of a slot $i$ to the front of the linked list pointed at by slot $i + 1$, and then swinging slot $i$’s pointer to point to the new front of the list. Details of the proofs as well as the formal statements of the step complexity bounds are presented in Appendix B.

5 The Lock Algorithm

We now present the lock algorithm, whose pseudocode is presented in Algorithm 2. Each tryLock attempt creates a descriptor, which specifies the list of locks to be acquired, the code to run if the locks are acquired successfully, and two other metadata fields: the priority assigned to this descriptor, and its current status. Initially, the priority is assigned the value $-1$, to indicate that the descriptor isn’t competing yet. The status is set to active initially, and can be changed to failed or won later in the execution as the fate of this attempt is determined.

After initializing its descriptor, a process starts its tryLock attempt with that descriptor. In a slight abuse of notation, we sometimes use a descriptor $p$ to refer to the process that initialized $p$ as it is executing the attempt tied to $p$. Without loss of generality we assume that each attempt
is tied to a unique descriptor. At a high level, the algorithm implements each lock as an active set object, using Algorithm 1 described in Section 4. A descriptor inserts itself into the active set of each lock, and then updates its priority to a value chosen uniformly at random. The descriptor then calls getSet on each of its locks in turn, and compares its priority to that of all other descriptors in the set. Intuitively, if a descriptor $p$ had the maximum priority of all descriptors on all of its locks, then it wins, and its thunk gets executed.

However, the algorithm is more subtle, as it must block the adversary from skewing the distribution of a given descriptor $p$’s competitors. Therefore, upon starting a new tryLock attempt, before inserting itself into any lock’s active set, and in particular, before choosing a random priority, $p$ helps all descriptors on its locks that have already revealed their priority. Intuitively, this is done to ‘clear the playing field’ by ensuring that any descriptor whose priority might have affected $p$’s adversarial start time cannot compete with $p$. To help other descriptors, $p$ executes a getSet on each of its locks in turn, and checks whether their priority is non-negative. For each descriptor $p'$ with a non-negative priority that $p$ sees, $p$ helps $p'$ determine whether it will win or lose. To do so, it calls the run function, which serves as the helping function and is the way that a descriptor competes against other descriptors. We describe the run function in more detail below; this function is the core of the lock algorithm. Before describing it, we first explain what a descriptor $p$ does after helping, and when it calls the run function to help itself.

After having executed the run function for every competitor that had a non-negative priority, it is time for $p$ to enter the tournament itself. First, $p$ inserts a pointer to itself into the active sets of all of its locks. After having done that, $p$ picks a random priority uniformly at random, and writes it into $p$.priority. We call this $p$’s reveal step, since it now reveals its priority to all other descriptors, and can now start receiving help from others. Note that since a pointer to $p$ is present in the active sets of all of its locks, any getSet on one of those locks that starts its execution after $p$’s reveal step will return $p$ and any reader will see $p$’s non-negative priority. $p$ now calls run(p) to compete in the tournament.

After returning from the run(p) call, $p$ is guaranteed to have a non-active status (either won or failed). That is, it knows the outcome of its attempt. At this point, $p$ cleans up after itself by removing itself from all active sets it was in.

**The run function.** The run function forms the core of the lock algorithm. The run function on a descriptor $p'$ checks the active sets of all of $p'$’s locks, and compares $p'$’s priority to all descriptors $q$ in those sets such that (1) $q$’s priority is non-negative, and (2) $q$’s status is active. On each such comparison, the descriptor with the lower priority is eliminated. This means that its status is atomically CASed from active to failed. After comparing $p'$’s priority with all descriptors in the active sets of all of its locks, the run function decides whether $p'$ won or lost. This involves trying to atomically CAS $p'$’s status to won. This will work if and only if $p'$ hasn’t been previously eliminated. Finally, run(p') ‘celebrates’ the end of $p'$’s competition by running its thunk if its status is won. The celebration-if-won is also executed for each competitor that $p'$ faced. This ensures that any descriptor that reaches the won status gets its thunk executed.

**Delays.** The algorithm as described thus far captures the essence of the approach; clear out any competitors whose priorities could have had an effect on your start time, and then compete by inserting yourself into the active sets of your locks and comparing your priority to all others. However, it also has weak points that the adversary can exploit to skew the priority distribution of the competitors of certain descriptors. In particular, a descriptor $p$ takes a variable amount of its own steps to get to its reveal point, and a variable amount of steps after that to finish its
struct Descriptor:
    lockList //list of active set objects
    thunk
    priority
    status = {active, won, lost}

tryLocks(lockList, thunk):
    p = new Descriptor(lockList, thunk, -1, active)
    for each lock ℓ in p.lockList
        set = ℓ.getSet()
        for each p' in set:
            if p'.priority != -1:
                run(p')
    for each lock ℓ in p.lockList
        ℓ.insert(p)
    Delay until $T_0$ total steps taken for constant k
    p.priority = rand //reveal step of p
    run(p)
    for each lock ℓ in p.lockList
        ℓ.remove(p)
    Delay until $T_1$ steps taken since previous delay

run(Descriptor p):
    for each lock ℓ in p.lockList
        set = ℓ.getSet()
        if (p.status == active):
            for p' in set:
                if (p'.status == active && p'.priority > -1):
                    if p.priority > p'.priority:
                        eliminate(p')
                    else if (p != p'): eliminate(p)
                celebrateIfWon(p')
            decide(p)
        celebrateIfWon(p)

decide(Descriptor p):
    CAS(p.status, active, won)

eliminate(Descriptor p):
    CAS(p.status, active, failed)

celebrateIfWon(Descriptor p):
    if(p.status == won):
        run p.thunk
attempt. This variance is caused by the amount of contention it experiences – how many descriptors are accessing the active set or are in it when \( p \) accesses the same active set, and what are their priorities. The number of other descriptors affect the time its insertion into the active sets takes (as shown in Section 1), as well as the number of descriptors it must compare its priority to. Furthermore, if \( p \) runs a descriptor’s thunk, this could take longer than if it simply eliminated it. The adversary can use this variance to skew the distribution of priorities of descriptors that \( p \) competes against.

To avoid this, we inject delays at two critical points in the algorithm. The first is immediately before \( p \)’s reveal step. The goal is to ensure that \( p \) always takes a fixed number of steps from its start time until its reveal step. This means that once the adversary chooses to start \( p \), it has also chosen its reveal time, and cannot modify this after discovering more information. To achieve this goal, we choose a fixed number of steps until \( p \)’s reveal step that is an upper bound on the amount of time \( p \) can take to arrive at its reveal step: \( T_0 = c \cdot \kappa^2 \cdot T \), where \( \kappa \) is the maximum point contention any descriptor can experience across all of its locks. This corresponds to the time it takes \( p \) to help all descriptors on each of its locks (up to \( \kappa \)), each of them having to compete with all descriptors on all of their locks (also up to \( \kappa \)). This includes the time \( T \) that it would take to run each of their thunks if they win. The \( c \) used in the delay expression is any sufficiently large constant to cover local instructions and the time for \( p \) to insert itself into all of its locks.

Similarly, we introduce a delay after \( p \)’s run to ensure that the time between its reveal step and termination is also determined at its invocation. Here, there is no need to square \( \kappa \), since \( p \) only needs to execute \text{run} for itself after its reveal step. Therefore, \( T_1 = c \cdot \kappa \) for some sufficiently large constant \( c \).

5.1 Safety and Fairness

In Appendix C, we present full proofs of the safety and fairness guarantees provided by the algorithm. Here, we give a brief overview of the key technical points.

We show that Algorithm 2 is correct by showing that it satisfies the mutual exclusion with idempotence property (Definition 2). The key to its correctness is in the way that the \text{run} function works. In particular, a descriptor’s status can change at most once. Furthermore, \text{celebrateIfWon} never actually runs a thunk unless it status is \text{won} (at which point it cannot lose anymore). Since its status can become \text{won} only in Line 33 after it compares its priority to that of the descriptors on all of its locks, and also celebrates any winners out of these descriptors, by the time it celebrates for itself, the thunks of any earlier winners on any of its locks have already been executed. Thus, the placements of the celebrations (once on Line 32 for its competitors, and once on Line 34 for itself) are crucial for the safety of the algorithm.

The fairness proof is more subtle. In essence, we show that the adversary’s power is quite limited. In particular, the adversary must decide whether or not two descriptors could threaten each other (i.e. their priorities could be compared in Line 28) before knowing any information on either of their priorities. Intuitively, this is due to two main reasons.

The first is because of the helping mechanism. Before a descriptor \( p \) reveals its priority, it puts all descriptors whose priority was already revealed in a state in which they can no longer threaten it – their status becomes non-active. The following lemma captures this property, where we say that a descriptor \( p \) causes another descriptor \( p' \) to fail if their priorities were compared on Line 28 and \( p \)’s priority was higher.

\begin{lemma}
Let \( p \) and \( p' \) be descriptors such that \( p \)’s \text{tryLock} starts after \( p' \)’s reveal step. Then neither descriptor can cause the other to fail.
\end{lemma}
That is, in this lemma we show that if \( p \) starts after \( p' \)'s reveal step, their priorities can never be compared.

The other crucial property of the algorithm that ensures that the adversary cannot decide to pit descriptors against one another after knowing their priorities stems from the delays injected into the algorithm. In particular,

**Observation 1.** Each descriptor interval takes the same number of steps by the initiating process between its start and its reveal step, and between its reveal step and the end of its interval, regardless of the schedule or randomness.

Together, these two properties allow us to prove the main lemma for the fairness the argument. This lemma relies on the notion of potential threateners. We define the set of the potential threateners of a descriptor \( p \) as the set of descriptors whose priorities might be compared to \( p \)'s priority in the execution. These are the descriptors whose tryLock intervals were ongoing when \( p \) reveals its priority and whose lock sets overlap \( p \)'s lock set (therefore they might appear in \( p \)'s active set), but which did not reveal their priorities before \( p \) started its interval (since by Lemma 5.1, these cannot harm \( p \)).

**Lemma 5.2.** The event that \( p \) is a potential threatener of \( p' \) in an execution \( E \) is independent of both \( p \) and \( p' \)'s priorities.

The final theorem is easily implied from this lemma by recalling that the choice of priorities of each descriptor is always done uniformly at random and independently of the history so far. Thus, the adversary can choose whether or not to introduce more threateners for a descriptor \( p \), but cannot affect their priorities. Since there is a bound on the amount of contention the adversary can introduce, we get a bound on \( p \)'s chance of success.

**Theorem 5.3.** Let \( C_\ell \) be the bound on the maximum point contention possible on lock \( \ell \), and let \( \kappa_p = \sum_{\ell \in p.lockList} C_\ell \) be the sum of the bounds on the point contention across all locks in a descriptor \( p \)'s lock list. Then the probability that \( p \) succeeds in its tryLock is at least \( \frac{1}{\kappa_p} \).

**Using the active set implementation.** We note that as shown by Golab et al. [23], using linearizable rather than atomic objects in a randomized algorithm can affect the probability distributions that an adversarial scheduler can produce. This effect can occur when several operations on the linearizable objects are executed concurrently. Thus, we must be careful when using our linearizable active set object in our randomized lock implementation. However, the way in which our lock algorithm uses the active set, and the way we use it in our analysis, is not subject to this effect. To see this, first note that an inserted descriptor in an active set is ignored until after its priority becomes non-negative, which happens after the insertion is complete. Similarly, a descriptor in an active set is ignored by the lock algorithm if its status is not active, which must happen before its removal begins. Finally, note that there is slack in our analysis; we consider the set of descriptors with potential to compete, where this means that a getSet executed by one descriptor could see the other descriptor with a non-negative priority. That is, any change in priority distributions that the adversary could try to achieve is already covered by our analysis.

We note that with negative-valued priorities and non-active statuses being skipped by the helping mechanism of our lock, we can see our use of the active set as a non-linearizable one. More specifically, if we consider ‘active’ only the elements with an active status and a non-negative priority, then the active set implementation we use is not linearizable. In particular, two getSet instances can be executed non-concurrently, and the later one can see some descriptor with a negative priority while the earlier one sees it with a non-negative priority. However, by the linearizability of the
base active set algorithm and the fact that a descriptor doesn’t change its priority or status more
than once, we guarantee a regular behavior with respect to these features; any getSet operation
that starts after a descriptor sets its priority to a non-negative value will see that priority value.
Similarly, any getSet that starts after a descriptor changed its status to a non-active value will see
the non-active status (or not see the descriptor at all). A similar correctness condition was defined
by Afek et al. [1] for their original definition of an active set object. Importantly, this condition is
all that is needed for the correctness and fairness of our lock algorithm.

5.2 Handling an Unknown Bound on Contention

So far, we’ve been assuming that \( \kappa \), the upper bound on the point contention of any tryLock
attempt, is known to the lock algorithm. We now address the case in which the bound \( \kappa \) exists,
is known to the adversary, but is unknown to the algorithm. We make slight modifications to the
locking algorithm, which are explained below. The pseudocode of the modified version is presented
in Algorithm 5 in Appendix D.

Algorithm 2 used the bound \( \kappa \) in two ways: firstly, the active set objects were instantiated with
arrays of size \( C \), where \( C \) was the maximum point contention on any single lock, and secondly, the
bound \( \kappa \) was used to determine how many delay steps each tryLock attempt must take to ensure
that each descriptor’s reveal step and final step of the attempt are always taken the same number
of steps after the attempt’s start time. The first concern is easy to fix by using more space; instead
of setting the size of the announcement array of the active set object to \( C \), we set it to \( P \), the total
number of processes in the system. In most applications, this number is significantly larger than
\( C \). Thus, we raise the size of the array to \( P \). We note that the size of each individual set pointed
at by the array slots will still be at most \( C \).

However, the second problem is more challenging. In particular, recall that the fixed times of
the reveal step and final step of each descriptor’s intervals were crucial in showing Lemma C.9,
that the priorities of two descriptors \( p \) and \( p' \) are independent of the event that \( p \) is a potential
threatener of \( p' \). This property was in turn crucial to get the desired fairness bound. Without
explicitly using \( \kappa \) to determine the amount of delay steps to take, it is unclear how to ensure that
the adversary doesn’t gain the power to choose whether two descriptors threaten each other based
on their priorities.

To handle an unknown \( \kappa \), we replace the two delay stages of the algorithm (Lines 16 and 21)
with different delaying strategies. Easier to handle is the delay at the end of the interval; note that
after the reveal step of a descriptor \( p \), the adversaries have two ways of controlling the number of
steps of the process executing \( p \): (1) the celebrations can take varying amounts of time depending
on the thunk length and whether or not the thunk is actually run, and (2) the size of the active
set read on line 25 can vary. Note however that if we can fix the size of the active sets read for
each lock, fixing the other problem is easy; since we still know \( T \), the maximum number of steps
that a thunk can take, we can ensure that for every descriptor read from the active set, we spend
\( T \) time, regardless of whether we really run its thunk. Therefore, we focus on how to fix the size of
the active sets.

For this purpose, we split the reveal step into two parts; the participation-reveal step occurs
after a descriptor \( p \) inserts itself into the active set object of each of its locks. In this step, it changes
its priority from \(-1\) to a special TBD value, indicating that it is ready to participate in the lock
competition, but not yet revealing its priority. This is the point at which we measure \( p \)'s potential
threateners in the analysis, and any descriptor starting after \( p \) reaches this point will help execution
\text{run}(p), and will not have \( p \) as a potential threatener. In the \text{run}(\hat{p}) function, all locks are queried
to obtain their active sets, and only then is the priority of \( p \) revealed. This time, since others may
help p execute this step, the priority is determined by using a CAS to change its value from TBD to a uniformly random value. This is called p’s priority-reveal step. The key insight is that after the priority is revealed, the active set objects are no longer queried, and instead the local copies of the sets, obtained just before the priority reveal step, are used. This means that the adversary does not learn p’s priority until after it can no longer affect the set of p’s potential threateners, and in particular, the remaining steps in p’s execution.

**Observation 2.** In Algorithm 5, the number of steps taken by the calling process of a descriptor p after the time $P_r$ at which p’s priority is revealed is fixed before $P_r$.

We now turn our attention to the steps a descriptor executes before its participation-reveal step. In the first part of its execution, a descriptor must help others to complete their run call, and must then insert itself into the active sets of all of its locks. The length of these tasks vary depending on the number of active descriptors in the system, and can therefore be controlled by the player adversary. For example, the adversary can employ the following strategy: let a descriptor p start its execution, and then let another descriptor $p'$ reveal its priority. If $p'$’s priority is above some threshold, let p reach its own reveal step quickly by preventing other descriptors from joining. Otherwise, if $p'$’s priority is low, inflate p’s time until its reveal step by introducing more descriptors. In this way, $p'$ could be made to threaten p if it’s priority is high, and not threaten p if it’s priority is low. Algorithm 5 employed fixed delays to get rid of this effect. However, the knowledge of the maximum bound on the contention, $\kappa$, was needed to determine how long to delay p.

Instead of relying on $\kappa$, we employ a doubling trick; p measures the number of steps it took until right before its participation-reveal step, and then employs a delay to bring that number up to the nearest power of two. In this way, while the adversary still has control of the number of steps p will take, this number is now guaranteed to be one of only $\log \kappa_p$ values, where $\kappa_p$ is the maximum contention p can experience across all of its locks in any execution ($\kappa_p$ differs from $\kappa$ if p uses less locks or its locks’ maximum contention is less than the maximum).

**Observation 3.** In Algorithm 5, the number of steps taken by the calling process of a descriptor p after the time $S_p$ at which p starts its attempt until its participation-reveal time is determined to be one of $\log \kappa_p$ fixed values before $S_p$.

Similarly to the proof of the lock algorithm with known contention bounds, we show that all threateners of a descriptor p must overlap its participation-reveal step. Intuitively, this is because any descriptor that overlaps p’s lock set and starts its execution after this step will necessarily help p (it must see p with a TBD priority when it executes getSet).

**Lemma 5.4.** Let p and $p'$ be descriptors such that p’s tryLock starts after $p'$’s participation-reveal step. Then neither descriptor can cause the other to fail.

The proof of this lemma is almost identical to the proof of its counterpart for the known-bound algorithm (Lemma 5.1).

**Observation 4.** The set of descriptors whose intervals overlap a descriptor p’s participation-reveal step includes all of p’s threateners.

We can now prove the main theorem, using a similar argument to proof of the fairness theorem of the known-bound version. The main difference is that here, a descriptor p’s participation reveal step is not completely predetermined at its start, but rather has $\log \kappa_p$ possible locations in the execution, where $\kappa_p$ is the maximum contention p can experience across its locks.
Theorem 5.5. Let $C_\ell$ be the bound on the maximum point contention possible on lock $\ell$, and let $\kappa_p = \sum_{\ell \in p.lockList} C_\ell$ be the sum of the bounds on the point contention across all locks in a descriptor $p$’s lock list. Then the probability that $p$ succeeds in its tryLock is at least $\frac{1}{\kappa_p \log \kappa_p}$ in the lock algorithm that does not know the bounds $C_\ell$ (Algorithm 5).

References

[1] Karl R. Abrahamson. On achieving consensus using a shared memory. In ACM Symposium on Principles of Distributed Computing (PODC), pages 291–302, 1988.

[2] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In ACM Symposium on Theory of Computing (STOC), pages 538–547, 1995.

[3] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. Disentangling multi-object operations (extended abstract). In ACM Symposium on Principles of Distributed Computing (PODC), 1997.

[4] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In IEEE Symposium on Foundations of Computer Science (FOCS), pages 262–272. IEEE, 1999.

[5] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. Concurrent deferred reference counting with constant-time overhead. In ACM Conference on Programming Language Design and Implementation (PLDI), 2021.

[6] Hagit Attiya and Eyal Dagan. Improved implementations of binary universal operations. J. ACM, 48(5), September 2001.

[7] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In ACM Symposium on Principles of Distributed Computing (PODC), 2006.

[8] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 336–343. ACM, 2008.

[9] Greg Barnes. A method for implementing lock-free shared-data structures. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 261–270, 1993.

[10] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 253–264, 2019.

[11] Guy E. Blelloch and Yuanhao Wei. LL/SC and Atomic Copy: Constant Time, Space Efficient Implementations Using Only Pointer-Width CAS. In International Symposium on Distributed Computing (DISC), 2020.

[12] Jeremy Brown, J. P. Grossman, and Tom Knight. A lightweight idempotent messaging protocol for faulty networks. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2002.

[13] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In ACM Symposium on Principles of Distributed Computing (PODC), 2013.
[14] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Scientific Computing*, 23(4), August 1994.

[15] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1), 1986.

[16] Marc de Kruifj and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[17] Marc A. de Kruifj, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[18] Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized dining philosophers without fairness assumption. In *Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP*, pages 169–180. Kluwer, 2002.

[19] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2010.

[20] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5–es, 2007.

[21] George Giakkoupis and Philipp Woelfel. A tight rmr lower bound for randomized mutual exclusion. In *ACM Symposium on Theory of Computing (STOC)*, pages 983–1002, 2012.

[22] George Giakkoupis and Philipp Woelfel. Randomized mutual exclusion with constant amortized rmr complexity on the dsm. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 504–513. IEEE, 2014.

[23] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *ACM Symposium on Theory of Computing (STOC)*, pages 373–382, 2011.

[24] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

[25] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.

[26] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[27] Peter Zilahy Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[28] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), June 1981.

[29] Eyal Kushilevitz and Michael O Rabin. Randomized mutual exclusion algorithms revisited. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 275–283, 1992.
In this section we describe a simulation that converts a thunk involving Read, Write and compare-and-modify (CAM) instructions into one that is idempotent, and prove Theorem 3.1. As the theorem states we do require a couple restrictions. Firstly we require that a thunk have a constant amount of local memory, to which we refer as its registers (these belong to the thunk independently of what process they run on). All mutable state beyond the registers must be accessed via Read, Write and CAM operations on memory locations. We also disallow concurrent Writes and CAMs to the same location.

In our simulation, each memory operation is simulated by some number of other operations. To help us discuss concurrent memory operations, we split each simulated memory operation into an invocation and response, and refer to the operations used to implement the simulation as primitive operations. A simulation is linearizable if, for each memory operation op in the history of a simulation, the effect of op happens between op's invocation and its response.

We base our simulation on two other operations, labeled load link (LLL) and labeled store conditional (LSC). An LLL takes a location and returns a "label" along with the value held at the memory location. An LSC takes a destination location loc, a label old, and a new value new. If old was the label returned by a LLL on loc, a later LSC on loc will succeed and write new to loc if and only if no successful LSC to loc linearized between the LLL and LSC. If old was not read from
loc, the behavior is undefined. Note that the the LLL and LSC are similar to standard load-linked (LL) and store-conditional (SC) operations, but differ in that the LSC need not be executed by the same process as the LLL. Also here we are assuming the LSC does not return whether it succeeded or not, unlike usually assumed for SC.

The LLL and LSC operations can be implemented using standard techniques. For example, an implementation can attach a counter to every memory location and when executing an LLL, return the counter as the label along with the value. The LSC can then be implemented by first checking if the counter is the same and getting the old value, and if the same executing a double word CAM conditionally replacing the old value and old-counter, with the new value and counter incremented by one. This involves a double-word CAM (or CAS), but an implementation could avoid both a double-word CAS and an unbounded counter by instead using a level of indirection. In this case the label is actually a pointer to the value, and the CAM just has to use the pointer for its old, and a new pointer to the new value for its new. This requires some form of constant-time safe memory reclamation to ensure a pointer is only reused after no other process has access to it [11].

Our simulation works in two stages. Firstly, we implement traditional Reads, Writes and CAMs using LLL and LSC, as shown in Figure 3. Secondly we simulate an LSC by adding a synchronization among processes. The idea of the synchronization is to have all processes working on a thunk agree on the state of the thunk before making any updates (LSCs). We define a context as the program counter and the current value of the registers for a thunk as it runs. A context can be used to represent the original thunk since the captured arguments (or pointers to them) can be kept in the registers, and the program counter would point to the first instruction of the thunk. We keep a copy of the context in memory, originally the initial thunk itself, and then periodically updated as the thunk proceeds. We refer to this as the shared thunk. Each process running a thunk also keeps its own copy of the most recent context it read from memory, which we call the previous context. We run the thunk as normal, except that before every CAM, we add a context update operation. The context update does a multiword (adjacent words) CAM on the shared thunk with the expected value being the previous context, and the new value being the current context. Note a multiword CAM can be implemented in constant time with a single-word CAM using indirection [11]. The purpose of the context update is to ensure that all runs of the thunk are in the same state before the LSC (what is important is that all LSC are applied with the same arguments). We refer to the instructions between two context updates as a capsule. The pseudocode is shown in Figure 4.

Without loss of generality we assume each context stored to the contextLocation is unique. If not, we can just add a capsule number to the context to make it unique.

Proof of Theorem 3.1. First we outline the correctness (linearizability) of the implementation of concurrent Reads and Writes, and Read and CAMs in terms of LLL and LSC. We do not allow concurrent writes and CAMs. A Read will always linearize at its LLL. An Write will linearize on the

```
read(x) { return LLL(x).val; }
write(x, y) {
    lv = LLL(x).label;
    LSC(x, lv.label, y);
}
CAM(x, old, new) {
    lv = LLL(x);
    if (new != old and lv.val == old) LSC(x, lv.label, new);
}
```

Algorithm 3: Simulating Read, Write and CAM using LLL and LSC.
Algorithm 4: Simulating code containing LLL and LSC so it is idempotent. The LLL is unmodified and the LSC uses the simulatedLSC.

LSC if successful, and otherwise linearize immediately before the write that corresponds to the first successful LSC after the LLL. There must be one before the Write’s LSC since it is not successful. This gives the correct behavior when interleaved with Reads since a Write will either appear to be immediately overwritten by a following successful Write, or will be properly written after its LSC. The CAM will linearize at the LLL if either of the conditions fail, and at the LSC otherwise. Again this gives the correct behavior when interleaved with Reads.

We consider the simulation based on LLLs and LSCs. We consider how the thunk progresses through its capsules, proving correctness by induction on $i$, where $i$ represents the $i$-th successful CAM of the context on Line 15 by any process helping with the thunk. In particular for $i$ we assume that the thunk is idempotent to that point, and the context written is consistent with a single process running on the thunk to that point. We refer to the $i$-th context as $c_i$. At this point there might be processes that are still working on capsule $i$ (the one before the $i$-th successful CAM) or even earlier capsules. Firstly we want to show that no LSCs from these earlier capsules can succeed. We note that if two calls are made to an LSC with equal arguments, the second always fails. This is because if the first LSC succeeds it falls between reading the label and the second causing the second to fail, and if the first fails some other LSC fell between reading the label and it, which also falls between reading the label and the second, again causing the second to fail. Now any LSC from an earlier capsule $j \leq i$ will have read the same context as a capsule that has now completed its LSC, and therefore had the same arguments for its LSC, and will fail.

Secondly we take the first process running capsule $i + 1$ that executes its LSC step and add it to $H'$ (from Definition 1). This is correct since it is the same instruction a single process running the thunk would take. Further LSCs on step $i + 1$ will fail since they have the same arguments.

Thirdly we want to ensure the next context written (at the end of capsule $i + 1$) is correct. This is because the first thing they all do is the LSC. Only the first can succeed, but immediately after the LSC they are all in the same state. This is where it is important that the LSC does not return whether it was successful or not. From this point until the end of the capsule the states

```javascript
// previousContext;
// sharedThunkLocation;

run():
    previousContext = Read(contextLocation)
    load previousContext into registers
    jump to program counter

runThunk(thunkPointer):
    sharedThunkLocation = thunkPointer
    run()

customLSC(loc, old, new):
    contextUpdate()
    LSC(loc, old, new)

simulatedLSC(loc, old, new):
    contextUpdate()
    LSC(loc, old, new)
```
might diverge since we are allowing for races with other procedures or thunks, but none of them are doing updates that are visible to each other, and at the end one will succeed on its CAM of a new context. That one is a valid run of just a single process for that capsule since the LSC, and we can add all its LLLs to \( H' \). We have thus shown our invariants hold after the \( i+1 \)-th successful CAM of the context.

Finally when the first process finishes (responds), all previous LCSs in \( H' \) have been completed.

\[ \square \]

A.1 Practical Considerations

Although the simulation is not particularly cumbersome, there is still overhead involved in the multiword CAM used in \texttt{contextUpdate}. Here we point out that there are several ways to reduce the number of \texttt{contextUpdates} (i.e., capsules). Importantly, any number of simulated LSCs on distinct locations can run in the same capsule, as long as they are all at the start of the capsule, i.e., immediately after the \texttt{contextUpdate} and before any LLLs. Also, any static data need not be written to the shared context. Many potential uses of our fair locks would require at most two capsules with only one or two registers saved across the boundary. This would be the case, for example, in locked operations on a stack or queue.

B Active Set Correctness and Step Complexity Proofs

B.1 Correctness

We now show that Algorithm [1] is a linearizable implementation of an active set object. Towards this goal, we first show some basic properties of the algorithm, which will then help us specify well-defined linearization points.

To ease the discussion of the algorithm’s correctness, we introduce some helpful terminology. We assume that elements do not get reinserted into the active set after being removed. This is without loss of generality, as we can assign a unique id to each element when it is inserted. We say the owner of a slot \( i \) at time \( t \) is the element in \texttt{announcements}[i].owner at time \( t \). Similarly, the set of slot \( i \) at time \( t \) is the set pointed at by the \texttt{announcements}[i].set field at time \( t \). We say an element \( e \) is current at time \( t \) if there is an \( i \) such that \( e \) is the owner of slot \( i \) at time \( t \). Note that at any time \( t \), there may be elements in the set field of a slot that are not current. Furthermore, the current owner of a slot at time \( t \) may be null. We let iteration \( \ell \) of a call to the \texttt{climb} function be an iteration of the loop in the \texttt{climb} function where \( j = \ell \).

\textbf{Lemma B.1.} The owner of a slot \( i \) never appears in the set of any slot \( j \) such that \( j > i \).

\textit{Proof.} Note that the set field of all slots only gets updated in the \texttt{climb} function. The lemma can be proven by a simple induction on the length of the execution: at the beginning of the execution, all sets are empty, and so the lemma holds. After that, note that iteration \( i \) of the \texttt{climb} function adds the owner of \( i \) to the set of slot \( i \), but not any slot \( j > i \). Thus, the lemma still holds after another iteration of the \texttt{climb} function. \( \square \)

\textbf{Lemma B.2.} At any point in time, the only current element that may appear in the set of a slot \( i \) but not in the set of slot \( i+1 \) is the owner of slot \( i \).

\textit{Proof.} We prove the lemma by induction on the number of times slot \( i \)'s set changes. The lemma trivially holds at the beginning of the execution, since the initial state of the set of all slots is empty. Assume that the lemma holds after slot \( i \)'s set changed at most \( k \) times. Consider the \( k+1 \)th time
it changes. Note that a set can only be changed by the CAS on Line 15 in the climb function. This CAS always puts in a new value that contains slot $i$'s owner plus slot $i+1$'s set. Thus, the lemma holds.

**Lemma B.3.** By the end of iteration $i$ of a climb call, an owner of slot $i$ that was current at some time after the beginning of this iteration is contained in slot $i$'s set.

**Proof.** Note that if one of the two CAS instances called in iteration $i$ of this climb call was successful, then the lemma holds. So, consider the case in which both CAS attempts of this iteration failed, and let $q$ be the process that executed this call. Let $R_1$ be the first time that this climb call read the set of slot $i$ on Line 9 and let $R_2$ be the second time. Furthermore, let $C_1$ and $C_2$ be the first and second time (respectively) that climb call executed a CAS on announcements[1].set.

Note that between $R_j$ and $C_j$, the algorithm reads the owner of slot $i$ and adds it to the new value for the subsequent CAS. If both CAS instances failed, then slot $i$'s set must have changed between $R_1$ and $C_1$, and against between $R_2$ and $C_2$. Note that slot $i$'s set changes only during the execution of a climb through a successful CAS operation. Consider the process, $p$, that executed the successful CAS operation between $R_2$ and $C_2$. $p$ must have read announcements[1].set after $R_1$, since otherwise its old value would have been outdated when it executed its CAS. Therefore, $p$ must also have read announcements[1].owner after $R_1$, and added this owner to its new value for its CAS. Therefore, $p$'s successful CAS placed an owner of slot $i$ in slot $i$'s set that was current at a time after $R_1$, and in particular, was current after the climb call of $q$ began.

**Lemma B.4.** Let $t$ be the time at which a call to climb($i$) terminates. There is a time $t' < t$ at which an owner of slot $i$ that was current at some time from the call to climb($i$) until $t'$ is contained in the list pointed at by announcements[0].set.

**Proof.** To prove the lemma, we show that after iteration $j$ of a climb($i$) call, the owner of every slot $k$ such that $j \leq k \leq i$ that was current at the beginning of the $k$th iteration or later is contained in the set of slot $j$. By Lemma B.3 immediately after iteration $k$, an owner of slot $k$ appears in slot $k$'s set. Furthermore, no current owner of any slot that was in the set of slot $k$ before iteration $k$ is lost by the end of the iteration. This is implied by Lemma B.2 which shows that the only current owner that can be in slot $k$'s set but not slot $k+1$'s set is the owner of $k$, and the fact that the $k$th iteration CASes in slot $k+1$'s set plus $k$'s owner into $k$'s set. Therefore, the owners get collected across iterations, so that at the end of iteration $j$ of a climb($i$) call, the owner of every slot $k$ such that $j \leq k \leq i$ that was current at the beginning of the $k$th iteration or later is contained in the set of slot $j$.

We can now define the linearization points of the operations.

- The **getSet** operation linearizes at the read of the pointer to the set, on Line 18.
- The **insert(p)** operation linearizes at the first successful CAS on announcements[0].set that contained $p$ in the new set value.
- The **remove(i)** operation linearizes at the first successful CAS on announcements[0].set after the remove operation began whose new value did not contain the descriptor erased from slot $i$ by this remove(i).

Note that since insert operations call climb on the slot at which they inserted an element, Lemma B.4 implies that there is a time during the interval of an insert operation at which the new element is present in announcements[0].set. Furthermore, recall that the set field is only ever modified with CAS operations. Therefore, the linearization point of an insert operation is well defined.
Similarly, since a remove operation also calls climb on the index at which it removed an element, again by Lemma B.4, there is a CAS operation during the remove operation’s interval that changed announcements[0].set to a set that does not contain the removed element. Thus, the remove operation’s linearization point is well defined as well. It is easy to see that these linearization points yield a correct sequential execution, since getSet reads announcements[0].set, and both the insert and the remove operations linearize when their effect becomes visible in announcements[0].set.

B.2 Step Complexity

We begin by showing that the step complexity indeed depends on the point contention. That is, we define the membership point contention at a time \( t, \kappa_t \), to be the number of elements in the active set plus the number of ongoing inserts. We claim that the step complexity of the insert operations depends on \( \kappa_t \) where \( t \) is a point during the insert operation’s interval.

**Lemma B.5.** If an insert operation’s successful CAS is on announcements[i].owner, then at some point \( t \) during its execution, \( \kappa_t \) was at least \( i + 1 \).

**Proof.** We prove this by induction on \( i \). Clearly, if an insert successfully CASes its new element into slot 0’s owner, then at the time at which it executed its CAS, there was at least 1 ongoing insertion – its own. Assume that the lemma holds for an insert operation that placed its new value in slot \( k \). Consider an insert operation \( op \) that placed its new value in slot \( k + 1 \). Consider the owners that \( op \) saw on slots \( 0 \) to \( k \) that made its CAS fail. Call those owners the critical owners.

By the induction hypothesis, for every slot \( j \leq k \), the insert operation of the critical owner of slot \( j \) had a point during its interval at which \( \kappa \) was at least \( j \). Let that point for slot \( j \) be \( t_j \). Consider the slot \( \ell \) whose point \( t_\ell \) is the latest of all such points. Then note that at \( t_\ell \), the insert operations of the critical owners of all slots \( j \leq \ell \leq k \) must already be pending, since otherwise their \( t_j \) points would have been after \( t_\ell \), contradicting the definition of \( t_\ell \). Furthermore, note that \( \kappa_\ell \) must include at least \( \ell + 1 \) elements that were not inserted into slots larger than \( \ell \), since we can build an execution that does not have those insertions and is indistinguishable to the process that inserted \( \ell \)’s critical owner. Therefore, \( t_\ell \) is a point during which \( \kappa \) was at least \( k + 1 \). If \( t_\ell \) is during \( op \)’s interval, then we are done, since \( op \) itself adds another one to that total.

Otherwise, if \( t_\ell \) is not in \( op \)’s interval, then it must have been before \( op \) started, since all critical insertions were complete by the end of \( op \)’s interval. In this case, all insertions of the critical owners that \( op \) encountered must have already started at the time at which \( op \) started, and since \( op \) saw these owners, their removals did not yet happen. Therefore, the invocation of \( op \)’s interval is a point at which \( \kappa \geq k + 2 \).

Note that the remove operation starts at the same slot as its corresponding insertion placed its element. Therefore, the remove operation traverses the same amount of slots as its corresponding insertion, which, by Lemma B.5, is proportional to the point contention \( \kappa \) during the insertion. If the active set is used as a membership tracker wherein the insert and remove operations signify the beginning and end of an operation in some external system (as it is used in our lock algorithm), the insert operation’s point contention is also a valid point contention for the overall operation in the external system.

We now briefly discuss the step complexity of the rest of the active set algorithm. Of interest is the climb function, which must make copies of the set fields of each slot. If we assume that it always makes deep copies, then we can consider each such copy to take time linear in the size of the set being copied. Since the sets never contain more than \( \kappa_t \) elements at any time \( t \), we can bound this time by the point contention as well. Therefore, the insert and remove operations take
$O(\kappa^2)$ steps. However, we can improve the time it takes to copy each set into the new one by implementing each set as a linked list, and consing the new element into the front of the list. More precisely, the set field of a slot $i$ points to the head of the linked list representing its set. When executing the $i-1$th iteration of a climb, a process creates a new node containing the owner of $i-1$ and makes it point to the head of slot $i$’s list. It then swings slot $i-1$’s set pointer to point to the newly added head. Notice that in this approach, the set of slot $i$ is not changed in iteration $i-1$, since the linked list starting at the pointer from slot $i$’s set field still does not contain the owner of $i-1$. Using this technique for maintaining and updating the sets, each iteration of the climb function takes constant time, and therefore, by Lemma B.5 the total time for an insert or remove operation is $O(\kappa)$.

The getSet takes constant time, since it simply returns a pointer to the set. Note that this is still an atomic read of the set, since the set itself is never changed; for any $i$, every time the announcements[i].set element is changed, its pointer is simply swung to a new location.

**Space Complexity.** The announcements array has $C$ slots, each of which can point to a set of at most $C$ elements. However, these $C$ elements are shared among the sets. In particular, each element only has one copy. Still, garbage collection must be employed to ensure the safe release of elements that have been removed from the set. This can be done using the wait-free reference counting presented in [5], yielding a total space usage of $O(C^2)$. If there are many active set objects in the same system (as is the case in our lock algorithm) the total space is $C$ per active set object in the system plus $O(C^2)$.

**C  Lock Algorithm Proofs**

**C.1 Safety Proof**

**Lemma C.1.** A descriptor’s status can change at most once during its execution.

*Proof.* Note that the status of a descriptor is only changed in the eliminate and decide functions, both of which use a CAS with an old value of active and if successful, change the status away from active. □

**Lemma C.2.** After a call to run(p), descriptor $p$ is not in the active status and if it is in the won status, its thunk has been run.

*Proof.* The decide(p) on Line 33 changes $p$’s status away from active. By Lemma C.1 once its status changes, it never changes again. Furthermore, celebrateIfWon(p) is called afterwards on Line 34 which, if $p$’s status is won, runs $p$’s thunk. □

**Lemma C.3.** If $p$ is in the failed state, its thunk is never run. Furthermore, a call to run p.thunk only happens after $p$ is in the won status.

*Proof.* A thunk is run only in a call to celebrateIfWon. The celebrateIfWon function does not run a thunk if the descriptor’s status is not won. □

**Lemma C.4.** If two calls run p.thunk and run p’.thunk are executed concurrently, neither descriptor had its thunk run to completion earlier, and the lock sets of $p$ and $p’$ overlap, then $p = p’$. 

24
Proof. Note that a descriptor’s status can only change to won in the call to decide(p) on Line 33 within a run(p) call, and that run(p) is only ever called after p’s priority is set to a non-negative value.

Assume by contradiction that p ≠ p', and let ℓ be a lock in the intersection of p.lockSet and p'.lockSet. By Lemma C.3 both p and p' must have reached the won status. Without loss of generality assume that p reaches the won status at time w_p, before p'. If p is in p’s set on Line 25 and p' celebrates for p after w_p, then p’s celebration, and thus the execution of its thunk, are completed before p’’s status is changed to won on Line 33. We consider two cases.

CASE 1. p’ celebrates for p before w_p. Then p’ must execute Line 28 for p before p’s reveal step, since otherwise p’ would have eliminated either itself or p, contradicting the assumption that they both eventually reach the won status. However, in this situation, since by assumption p reaches the won status first, p must have p’ in its set when it executes Line 25 and must have seen a non-negative priority for p’ and p’’s status as active. Therefore, p would have eliminated one of the two, again contradicting our assumption that they both eventually execute their thunks.

CASE 2. p’ does not have p in its set in Line 25. Then by the correctness of the active set algorithm, p must not have inserted itself into the set yet. Similarly to the argument above, this means that p must get p’ in its set when it executes Line 25 and will observe p’ with a non-negative priority and an active status. Therefore, p would eliminate one of the two descriptors, again contradicting the assumption that they both eventually execute their thunks.

Therefore, the mutual exclusion property is maintained, and a thunk is run if and only if a descriptor wins its tryLock.

**Theorem C.5.** Algorithm 2 satisfies the mutual exclusion with idempotence property (Definition 2). Furthermore, if a tryLock attempt fails, then the corresponding thunk was never executed.

C.2 Fairness Proof

We say a descriptor p causes a descriptor p’ to fail if p’’s status is changed to failed in the eliminate(p’) call on Line 30 or 31 after comparing p’’s priority with p’s priority. We say a descriptor p can cause p’ to fail if p and p’’s priorities are compared on Line 30 or 31 during the execution.

**Lemma C.6.** A descriptor p cannot cause a descriptor p’ to fail if their lock sets do not intersect.

*Proof.* A descriptor p is only inserted into the sets corresponding to locks in its lock set (Line 14). Furthermore, in run(p), only the sets of locks in p’s lock sets are compared against. Therefore, a descriptor p’ will never be compared against and potentially eliminated by a descriptor p if their lock sets do not intersect. □

**Lemma C.7.** A descriptor p cannot cause a descriptor p’ to fail if p’s status stopped being active before p’’s reveal step.

*Proof.* First note that before p’’s reveal step, no descriptor will eliminate p’, since its priority will be negative, and the comparison with it will be skipped on Line 28. Therefore, p cannot cause p’ to fail before p’’s reveal step. By Lemma C.1 p’’s status will never be active again after it stops being active. Therefore, in any run(p) or run(p’) call, the comparison of p with any other descriptor will be skipped on Line 28 after p becomes inactive. In particular, this comparison will always be skipped after p’’s reveal step. □
Lemma C.8. Let \( p \) and \( p' \) be descriptors such that \( p \)'s tryLock starts after \( p' \)'s reveal step. Then neither descriptor can cause the other to fail.

Proof. By Lemma [C.6], if \( p \) and \( p' \)'s lock sets do not overlap, the lemma holds. Otherwise, if \( p' \) has already removed itself from the active set by the time \( p \) started its getSet on Line 10 then \( p' \) must have already won or failed by this time. In particular, \( p \) wasn’t in the active set during \( p' \)'s run\((p')\), and therefore could not have caused \( p' \) to fail. Furthermore, by Lemma [C.7] \( p' \) cannot cause \( p \) to fail.

So, assume that \( p' \) is still in the active set at the time \( p \) started its getSet on Line 10. By the linearizability of the active set algorithm, since \( p' \) must have completed its insertion into the active set before \( p \) started its getSet on Line 10 \( p \) must have \( p' \) in the set it gets. Furthermore, since \( p' \) already revealed its priority before \( p \)'s getSet linearized, \( p \) must see a non-negative priority for \( p' \). Therefore, \( p \) calls run\((p')\) before its own reveal step, so by Lemma [C.2] \( p' \) wins or fails because of a different descriptor. Furthermore, again by Lemma [C.7] \( p' \) cannot cause \( p \) to fail, since a run\((p')\) call completes before \( p \)'s reveal step, and therefore \( p' \) is no longer active by that time.

We define the interval of a descriptor \( p \) as the time between its calling process’s call to tryLock and the time at which its tryLock call returns. Furthermore, a descriptor’s threat interval is the time between the beginning of its interval and the time at which its status stops being active. We define \( p \)'s threateners as the set of descriptors that can cause \( p \) to fail.

Observation 5. The set of descriptors whose intervals overlap a descriptor \( p \)'s reveal step includes all of \( p \)'s threateners.

Proof. Every descriptor \( p \)'s interval includes a complete call to run\((p)\). Thus, by Lemma [C.2] the status of any descriptor is not active by the end of its interval. The lemma is therefore immediately implied from Lemmas [C.7] and [C.8].

Observation 6. At the time at which a descriptor’s interval starts, none of its threateners have reached their reveal step.

Proof. This is immediate from Lemma [C.8].

Observation 7. Each descriptor interval takes the same number of steps by the initiating process between its start and its reveal step, and between its reveal step and the end of its interval, regardless of the schedule or randomness.

Proof. This property is enforced by the delays injected in Lines 16 and 21.

We say that a descriptor \( p \) is a potential threatener of another descriptor \( p' \) if (1) \( p \)'s interval overlaps with \( p' \)'s reveal step, and (2) \( p' \) did not execute a run\((p)\). Note that by Observation 5 and Lemma [C.7] the set of potential threateners of a descriptor is a superset of its actual threateners.

Lemma C.9. The event that \( p \) is a potential threatener of \( p' \) in an execution \( E \) is independent of both \( p \) and \( p' \)'s priorities.

Proof. By Observation 7, once a descriptor starts, its reveal step and last step of its interval are determined. Since these two steps are what determines whether a descriptor will be a potential threatener of another descriptor, the start times of the two descriptors determine whether this event occurs. Furthermore, by Lemma [C.8] and the definition of potential threateners, if \( p \) is a potential threatener of \( p' \) in an execution \( E \), then both \( p \) and \( p' \) must have started their tryLock interval before either of their reveal steps. Therefore, the player adversary did not have any information on their priorities at the time at which it decided to start their intervals. Therefore the event that \( p \) is a potential threatener of \( p' \) is independent of both of their priorities.
Theorem C.10. Let $C_\ell$ be the bound on the maximum point contention possible on lock $\ell$, and let $\kappa_p = \sum_{\ell \in p.\text{lockList}} C_\ell$ be the sum of the bounds on the point contention across all locks in a descriptor $p$’s lock list. Then the probability that $p$ succeeds in its tryLock is at least $\frac{1}{\kappa_p}$.

Proof. On each lock $\ell$ in $p$’s lock list, the adversary can make at most $C_\ell$ descriptors be potential threateners of $p$. Assume that all priorities of the descriptors are picked uniformly at random at the beginning of the execution, but the priority of a given descriptor $p'$ is hidden until after the adversary chooses whether or not $p'$ will be a potential threatener of $p$. This is equivalent to our setting since the priorities are always picked uniformly at random, and, by Lemma C.9, the adversary has no information on a descriptor $p'$’s priority until after it decides whether it will potentially threaten $p$. Once the adversary discovers the priority of a descriptor, it can decide whether the next descriptor will be a potential threatener of $p$ and then reveal the corresponding priority. In the worst case, the adversary can reveal up to $\kappa_p$ of those predetermined priorities. Since the set of potential threateners of $p$ include all actual threateners of $p$, this makes $p$ threatened by $\kappa_p$ uniformly chosen random values in the worst case, giving it a $\frac{1}{\kappa_p}$ probability of having the maximum priority of all of them.

D Unknown Bound on Contention

Here we present the pseudocode for the version of the lock algorithm that does not know the contention bound $C$. The lines in red are the ones that changed over the original version presented in Algorithm 2.

E Proof

Lemma E.1. Let $p$ and $p'$ be descriptors such that $p$’s tryLock starts after $p'$’s participation-reveal step. Then neither descriptor can cause the other to fail.

The proof of this lemma is almost identical to the proof of its counterpart for the known-bound algorithm (Lemma 5.1).

Proof. By Lemma [C.6] if $p$ and $p'$’s lock sets do not overlap, the lemma holds. Otherwise, if $p'$ has already removed itself from the active set by the time $p$ started its getSet on Line 10 then $p'$ must have already won or failed by this time. In particular, $p$ wasn’t in the active set during $p'$’s run($p'$), and therefore could not have caused $p'$ to fail. Furthermore, by Lemma [C.7] $p'$ cannot cause $p$ to fail.

So, assume that $p'$ is still in the active set at the time $p$ started its getSet on Line 10. By the linearizability of the active set algorithm, since $p'$ must have completed its insertion into the active set before $p$ started its getSet on Line 10 $p$ must have $p'$ in the set it gets. Furthermore, since $p'$ already revealed its participation (set its priority to TBD) before $p$’s getSet linearized, $p$ must see a non-negative or TBD priority for $p'$. Therefore, $p$ calls run($p'$) before its own participation-reveal step, so by Lemma [C.2] $p'$ wins or fails because of a different descriptor. Furthermore, again by Lemma [C.7] $p'$ cannot cause $p$ to fail, since a run($p'$) call completes before $p$’s reveal step, and therefore $p'$ is no longer active by that time.

Observation 8. The set of descriptors whose intervals overlap a descriptor $p$’s participation-reveal step includes all of $p$’s threateners.
struct Descriptor:
  lockList //list of active set objects
  thunk
  priority
  status = {active, won, lost}

p = new Descriptor(lockList, thunk, -1, active)
tryLocks(p)

tryLocks(Descriptor p):
  for each lock ℓ in p.lockList
    set = ℓ.getSet()
    for each p’ in set:
      if p’.priority != -1:
        run(p’)
    for each lock ℓ in p.lockList
      ℓ.insert(p)
  Delay until total number of steps is the next power of two
  p.priority = TBD //participation reveal step
  numContenders = run(p)
  for each lock ℓ in p.lockList
    ℓ.remove(p)
  Delay until numContenders · T steps taken since previous delay

int run(Descriptor p):
  Lock[] sets
  numContenders = 0
  for each lock ℓ in p.lockList
    sets[L] = ℓ.getSet()
    numContenders += size(sets[L])
  CAS(p.priority, TBD, rand) //priority reveal step
  for each lock ℓ in p.lockList
    if (p.status == active):
      for p’ in sets[ℓ]:
        if (p’.status == active && p’.priority > -1):
          if p.priority > p’.priority:
            eliminate(p’)
          else if (p != p’): eliminate(p)
          celebrate(p’)
        decide(p)
      celebrate(p)
    return numContenders

decide(Descriptor p):
  CAS(p.status, active, won)

eliminate(Descriptor p):
  CAS(p.status, active, failed)

celebrate(Descriptor p):
  if(p.status == won):
    run p.thunk

Algorithm 5: Lock Algorithm
Proof. Every descriptor $p$’s interval includes a complete call to $\text{run}(p)$. Thus, by Lemma [C.2] the status of any descriptor is not active by the end of its interval. The lemma is therefore immediately implied from Lemmas [C.7] and [C.8].

We can now prove the main theorem, using a similar argument to proof of the fairness theorem of the known-bound version. The main difference is that here, a descriptor $p$’s participation reveal step is not completely predetermined at its start, but rather has $\log \kappa_p$ possible locations in the execution, where $\kappa_p$ is the maximum contention $p$ can experience across its locks.

**Theorem E.2.** Let $C_\ell$ be the bound on the maximum point contention possible on lock $\ell$, and let $\kappa_p = \sum_{\ell \in p.\text{lockList}} C_\ell$ be the sum of the bounds on the point contention across all locks in a descriptor $p$’s lock list. Then the probability that $p$ succeeds in its tryLock is at least $\frac{1}{\kappa_p \log \kappa_p}$ in the lock algorithm that does not know the bounds $C_\ell$ (Algorithm 5).

Proof. At each of $p$’s $\log \kappa_p$ possible participation-reveal times, the adversary can make at most $\kappa_p$ descriptors be potential threateners of $p$. Assume that the priority of each descriptor $p'$ is picked uniformly at random at the beginning of the execution, but is hidden until after $p'$’s priority-reveal time. This matches our setting since priorities are always picked uniformly at random at the priority-reveal time. By Observation 2 the number of steps that a descriptor $p'$ takes after its priority-reveal time until it terminates is determined before $p'$’s priority-reveal time. Furthermore, note that the priority-reveal time of each of a descriptor $p$’s potential threateners must be after $p$ starts, by Lemma [E.1]. Additionally, by this time, $p$’s participation-reveal time is fixed to be at one of $\log \kappa_p$ steps in the execution. Thus, before knowing $p'$’s priority, the adversary must decide which of $p$’s possible participation-reveal times $p'$’s interval will overlap. Once the adversary decided this, it can reveal $p'$’s priority, and based on this, can decide whether to reveal other descriptors’ priorities. However, by the arguments above, there are at most $\kappa_p \log \kappa_p$ potential threateners of $p$ whose priorities it can reveal, and these priorities are all chosen uniformly at random. Therefore, $p$’s chance of success is at least the chance that its uniformly random priority is the higher of $\kappa_p \log \kappa_p$ uniformly random i.i.d. values – $\frac{1}{\kappa_p \log \kappa_p}$. □