UNIVERSITY OF CALGARY

RMR-Efficient Randomized Abortable Mutual Exclusion

by

Abhijeet Pareek

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

January, 2012

Abstract

Mutual exclusion [1], also known as locking, is a fundamental and well studied problem in distributed computing. In this problem, N processes synchronize among themselves to access a *Critical Section* such that no two processes access the Critical Section at the same time. Recent research on mutual exclusion locks for the asynchronous shared-memory model has focused on *local spin* algorithms that use the remote memory references (RMRs) metric.

There exists a $\Omega(\log N)$ lower bound by Attiya, Hendler and Woelfel [2] on the number of RMRs incurred by processes as they enter and exit the Critical Section, which matches an upper bound by Yang and Anderson [3]. The lower bound applies for deterministic algorithms that only use read and write operations, and there exists a randomized algorithm by Hendler and Woelfel [4], that only uses read and write operations, and achieves sub-logarithmic expected RMR complexity, against an *adaptive adversary* [5].

Local spin mutual exclusion locks do not meet a critical demand of many systems [6]. Specifically, the locks employed in database systems and in real time systems must support a "timeout" capability, that is, it must be possible for a process that waits "too long" to abort its attempt to acquire the lock. In real time systems, the abort capability can be used to avoid overshooting a deadline. Locks that allow a process to abort its attempt to acquire the lock are called abortable mutual exclusion locks. Jayanti presented an efficient deterministic abortable mutual exclusion lock [7] with worst-case $\mathcal{O}(\log N)$ RMR complexity.

The problem of designing a randomized abortable mutual exclusion algorithm with sublogarithmic expected RMRs has remained open. In this thesis, we solve this open problem by presenting a randomized abortable mutual exclusion with $\mathcal{O}(\log N/\log\log N)$ expected RMRs, against a *weak adversary* [5].

Acknowledgements

First and foremost I would like to thank my supervisor, Dr. Philipp Woelfel, for being Superman. Specifically I would like to thank him for his insight, supervision, patience and the constant guidance that he provided during the writing of my thesis. He reviewed my work tirelessly and provided a constructive and critical perspective that enabled me to write my thesis. His guidance served to be a tremendous learning experience for me, and I realized that solving a problem is only the first step, and writing up one's work unambiguously and precisely is as difficult and important a problem. Philipp taught me innumerable things during the course of my Masters and I will cherish my interactions with him for a long time to come.

I would like to thank my other committee members, Dr. Lisa Higham and Dr. Bill Sands for their time and their valuable feedback on my work that has helped shape this thesis into its current form.

I would like to thank my wonderful family, specially my dad, for their endless support and encouragement. Finally I would like to thank my partner in crime Shambhavi Srinivasa for going through the process of graduate school with me, for sharing every little success and failure of the process, and for always being there for me. Finishing my Masters without her love and support would have been a gazillion times harder.

Table of Contents

List of Figures

Chapter 1

Introduction

Multiprocessor systems are inherently *asynchronous*: processor activities can be paused or delayed without warning by system events such as interrupts, preemptions, cache misses, failures, etc. These delays are unpredictable, and can vary significantly in duration. In multiprocessor programming a fundamental and well studied problem is that of coordinating access to shared resources, while taking the asynchrony of the system into consideration. The most popular approach in practice for allowing multiple processes to access a shared resource safely is *mutual exclusion* [1], also known as locking. An algorithm for this problem implements a mutual exclusion lock, an object designed to be used by at most N processes at any time, and with the property that at most one process "owns" it at any time. A process is said to own a lock if it participates in a "capture" protocol designed for the object, and completes it. The owner of the lock can access the shared resource, while all other processes wait in their capture protocol for the owner to "release" the lock. The owner releases the lock by executing a release protocol designed for the object. The algorithm is required to satisfy some additional properties called *progress* properties, which assure that some participating process makes some progress towards capturing the lock. The weakest of the progress properties is *deadlock freedom*. Deadlock freedom assures that if all participating processes continue to take steps, then some process will capture the lock. Mutual exclusion locks are used extensively in operating systems and in asynchronous parallel applications to implement shared data structures.

Early mutual exclusion locks were designed for uniprocessor systems that supported multitasking and time-sharing. A comprehensive survey of these locking algorithms is presented in [8]. One of the biggest shortcomings of these early locking algorithms is that they did not take into account an important hardware technology trend – the steadily growing gap between high processor speeds and the low speed/bandwidth of the processor-memory interconnect [9]. A memory access that traverses the processor-to-memory interconnect, called a *remote memory reference*, takes much more time than a local memory access.

Not surprisingly, recent research [10, 11, 12, 13, 14, 2, 15, 7, 16] on locking algorithms is focused on minimizing the number of remote memory references (RMR). The maximum number of RMRs that any process requires to capture and release a lock is called the RMR complexity of the lock. RMR complexity is the metric used to analyze the efficiency of mutual exclusion algorithms, as opposed to the traditional metric of counting steps taken by a process (step complexity). Step complexity is problematic, since for mutual exclusion algorithms, a process may perform an unbounded number of memory accesses (each considered a step) while busy-waiting for another process to release the lock [17].

Algorithms that perform all busy-waiting by repeatedly reading *locally accessible* shared variables, achieve bounded RMR complexity and have practical performance benefits [18]. Such algorithms are termed *local spin* algorithms. A comprehensive survey of these algorithms is presented in [19]. Yang and Anderson presented the first $\mathcal{O}(\log N)$ RMRs mutual exclusion algorithm [3] using only reads and writes. Anderson and Kim [12] conjectured that this was optimal, and the conjecture was proved by Attiya, Hendler, and Woelfel [2].

Local spin mutual exclusion locks do not meet a critical demand of many systems [6]. Specifically, the locks employed in database systems and in real time systems must support a "timeout" capability. That is, it must be possible for a process that waits "too long" to abort its attempt to acquire the lock. The ability of a thread to abort its lock attempt is crucial in data base systems; for instance in Oracle's Parallel Server and

IBM's DB2, this ability serves the dual purpose of recovering from transaction deadlock and tolerating preemption of the thread that holds the lock [6]. In real time systems, the abort capability can be used to avoid overshooting a deadline. Locks that allow a process to abort its attempt to acquire the lock are called abortable mutual exclusion locks. Jayanti presented an efficient deterministic abortable mutual exclusion lock [7] with worst-case $\mathcal{O}(\log N)$ RMR complexity, where N is the number of processes that can access the lock concurrently.

Due to the inherent asynchrony in the system, the RMRs incurred by a process during a lock capture and release depend on how the steps of all the processes in the system were scheduled one after the other. Therefore, the maximum RMRs incurred by any process during any lock attempt are determined by the "worst" schedule that makes some process incur a large number of RMRs. To analyze the RMR complexity of lock algorithms, we define an adversarial scheduler called the *adversary*. The lower bound of $\Omega(\log N)$ in [2] for mutual exclusion algorithms that use only reads and writes holds for deterministic algorithms where the adversary knows all processes' future steps. The lower bound does not hold for randomized algorithms where processes flip coins to determine their next steps. Randomized algorithms limit the power of an adversary since the adversary cannot know the result of future coin flips. In literature, adversaries of varying powers are considered, namely the *oblivious*, *weak*, and *adaptive* adversary [5]. The adaptive adversary models the strongest adversary with reasonable powers. Hendler, and Woelfel [4] presented the first randomized mutual exclusion algorithm with $\mathcal{O}(\log N/\log\log N)$ expected RMR complexity, against the adaptive adversary, showing that randomization breaks the logarithmic barrier of [2].

In this thesis, we present a randomized abortable mutual exclusion lock, with $\mathcal{O}(\log N/\log\log N)$ expected RMR complexity against the weak adversary.

Chapter 2

Preliminaries

2.1 Asynchronous Shared Memory Model

Our model of computation, the asynchronous shared-memory model, is based on Herlihy and Wing's [20]. The asynchronous shared-memory system has a fixed set $\mathcal{P}$ of N processes which communicate via a set of globally *shared objects.* Every process executes its program by taking *steps*, and does not fail. A step is defined to be the execution of all local computations followed by an operation on a shared object. Every process takes steps until it has no more steps to take, at which point it is said to *terminate.* A process is *active* if is has not terminated.

A process *accesses* shared objects by applying operations on them. An operation is either *atomic* or *non-atomic.* An atomic operation happens instantaneously, and is modeled as a single step. A non-atomic operation is not instantaneous, and is modeled using separate invocation and response steps, which are executed in that order. We say that a response step *matches* an invocation step if the two steps are applied by the same process to the same shared object using a non-atomic operation.

Histories. A sequence of steps executed by processes is called a *history.* Sometimes a history is also referred to as an *execution.* A history is generated by recording every step executed by all processes on all base objects and all implemented objects. A *completed* operation is one whose invocation step and response step is present in a history, whereas a *pending* operation is one whose response step is absent from a history. We say that an operation *op happens before* op' in history H, if the response of op occurs before the invocation of op' in H. We say that op and op' are *concurrent* in H if neither happens before the other in H. A history H is said to be *sequential* if it only contains only atomic steps, or if no two operations in H are concurrent.

Objects, Object Types and Validity. An object J has a *type* $\mathcal{T} = (\mathcal{S}, s_0, \delta, O, \mathcal{R})$ where $\mathcal{S}$ is a set of states, $s_0 \in \mathcal{S}$ is the initial state, O is a set of operations, $\mathcal{R}$ is the set of responses, and $\delta : \mathcal{S} \times O \rightarrow \mathcal{S} \times \mathcal{R}$ is a many-to-one mapping. The mapping δ defines how operations are applied on object J. Specifically, if a process applies operation $op \in O$ to an object J of type $\mathcal{T}$ that is in state s, then J returns to the process a response r and changes the state of J to s' if and only if $\delta(s, op) = (s', r)$. An *object subhistory* $H|x$ of a history H is the subsequence of all steps in H executed on object x. A *sequential specification* of an object is the set of all possible sequential histories for that object. A sequential history S is *valid*, if for each object x, $S|x$ is in the sequential specification of x.

Implementations of shared objects. Given a set $\mathcal{J}$ of base objects of specified types, we can implement an object of a target type $\mathcal{T}$. An *implementation* defines how to simulate an object of type $\mathcal{T}$, using base objects $\mathcal{J}$. Formally, an implementation of an object of type $\mathcal{T}$ is denoted by the tuple $I = (\mathcal{P}, \mathcal{J}, \mathcal{H}, \mathcal{T})$ where $\mathcal{P}$ is the set of processes, $\mathcal{J}$ is the set of base objects, $\mathcal{H}$ is the set of histories, $\mathcal{T}$ is the target object type. The set $\mathcal{H}$ consists of histories generated by recording an invocation or response step on the target object whenever a process begins or finishes executing a method (respectively), and an atomic step for each access to a base object in $\mathcal{J}$.

In this thesis, we specify an implementation using pseudo-code to define a method for each operation type ot of the target type. The pseudo-code establishes how process p applies operations on the base objects, in order to apply an operation of type ot to the target object.

Linearizability. The gold standard in correctness conditions of an implemented shared object in concurrent systems is linearizability [20]. Informally, linearizability states that each operation executed on the target object appears to take effect instantaneously at some point between the operation's invocation and response (or possibly not at all if the operation execution is pending).

Definition 2.1.1. *Let H be a history and S a sequential history. We say that S is a linearization of H, if*

(a) S contains all completed operations of H, and possibly some uncompleted operations with responses appended with arbitrary return values,

(b) S is valid, and

(c) if operation op happens before op′ in H then op′ does not occur before op in S.

History H is linearizable, if it has a linearization. An implementation $I = (\mathcal{T}, \mathcal{P}, \mathcal{J}, \mathcal{H})$ is linearizable, if every history $H \in \mathcal{H}$ can be linearized to a sequential history in the sequential specification of $\mathcal{T}$.

Point Contention. We first establish a notion of time for an execution E, by saying that the i-th step in E occurs at time i. The *execution interval* of an operation *op* is the interval that begins at the time of *op*'s invocation step and ends at the time of *op*'s response step. Let O be the object on which the operation *op* is being applied. The *point contention* of *op* is the maximum number of operations on O that are pending at any point during *op*'s execution interval.

Wait-freedom, Bounded Wait-freedom and Lock-freedom. A section of code is said to be *wait-free* if a process can complete it in a finite number of its own steps. A section of code is said to be *bounded* if there is a bound on the number of steps required to completely execute the section of code. An object implementation is said to be wait-free if all its operations are wait-free. An object implementation is *lock-free*, if in any infinite history H where processes continue to take steps, and H contains only operations on that object, some operation finishes.

2.2 Shared Memory System Architectures

In this thesis, we consider two asynchronous shared memory computation models based on the *cache-coherent* (CC) and the *distributed shared memory* (DSM) multiprocessor architectures. In each architecture, a memory location is either *local* or *remote.*

CC Model. In the CC model, each processor has a private cache in which it maintains local copies of shared objects that it accesses. The private cache is logically situated "closer" to the processor than the shared memory, and is considered local to the processor. The shared memory is an external memory accessible to all processors, and is considered remote to all processors. We assume that a hardware protocol ensures cache consistency (i.e., that all copies of the same object in different caches are valid and consistent).

DSM Model. In the DSM model, each processor has its own memory module which is available locally and can be accessed without traversing the processor-to-memory interconnect. The memory module of a processor is considered local to the processor, and remote to all other processors. The shared memory is the disjoint union of all memory modules of all the processors in the system.

Remote Memory Reference (RMR). A memory access to a shared object that requires access to remote memory is called a *remote memory reference.* The *RMR complexity* of a shared memory algorithm is the maximum number of RMRs that a process can incur during any execution of the algorithm.

2.3 Primitive Objects

In this section we describe the three primitive objects of the asynchronous shared memory model, namely read-write registers, `CAS` objects and `LL/SC` objects. Most implementations of more sophisticated objects use these objects as the base objects in their implementations, since most modern architectures support either `CAS` and read-write registers (e.g., UltraSPARC [21] and Itanium [22]) or `LL/SC` and read-write registers (e.g., POWER [23] , MIPS [24] and Alpha [25]).

Read-Write Register. The simplest shared object in our model is a read-write register. A read-write register R stores a value from some set and supports two atomic operations R.`Read()` and R.`Write()`. Operation R.`Read()` returns the value of the register and leaves its content unchanged, and operation R.`Write(`v`)` writes the value v into the register and returns nothing. If the set of values that can be stored in the register is unbounded then the register is *unbounded*; otherwise the register is *bounded*.

LL/SC Object. An `LL/SC` object O stores a value from some set and supports two atomic operations O.`LL()` and O.`SC()`. Operation O.`LL()` returns the value stored in O. Operation O.`SC(`v`)` by a process p must follow the execution of an O.`LL()` operation by p, and the operation may *succeed* or *fail*. A successful O.`SC(`v`)` changes the value of O to v and returns **true**, otherwise the value of O is unchanged and **false** is returned. Operation O.`SC(`v`)` is successful, if and only if no process has performed a successful `SC()` operation on O since the execution of p's preceding `LL()` operation on O.

CAS Object. A `CAS` object O stores a value from some set and supports two atomic operations O.`CAS()` and O.`Read()`. Operation O.`Read()` returns the value stored in O. Operation O.`CAS(`exp, new`)` takes two arguments exp and new and attempts to change the value of O from exp to new. If the value of O equals exp then the operation

O.`CAS`(exp, new) *succeeds*, and the value of O is changed from *exp* to *new*, and **true** is returned. Otherwise, the operation fails, and the value of O remains unchanged and **false** is returned.

Equivalence of LL/SC, CAS and Read-Write Registers for Deterministic Algorithms. It turns out that an `LL/SC` object has an efficient implementation (constant RMR and constant space per process) from `CAS` objects and vice-versa [26]. Golab, Hadzilacos, Hendler, and Woelfel [27] (see also [28]) presented an $\mathcal{O}(1)$-RMRs implementation of a `CAS` object using only read-write registers. Moreover, they proved that one can simulate any deterministic shared memory algorithm that uses reads, writes, and conditional operations (such as `CAS` operations), with a deterministic algorithm that uses only reads and writes, with only a constant increase in the RMR complexity.

2.4 Adversary Models for Randomized Distributed Algorithms

A randomized algorithm is an algorithm where processes, at times, randomly choose the next step of the algorithm. The randomness in the algorithm is modeled by coin flips. Processes can perform independent random experiments by executing `flip()` operations (with no arguments) on a single shared coin object. Flip operations on a coin object are assumed to be atomic, and return a value from an arbitrary countable set Ω, called the coin flip *domain.* A vector $\vec{c} = (c_1, c_2, \ldots) \in \Omega^\infty$ is called a *coin flip vector.* A history H is said to *observe* the coin flip vector $\vec{c} = (c_1, c_2, \ldots)$, if the i-th flip operation in H returns value c_i.

A *schedule* is the order in which steps of processes interleave, represented by a sequence (possibly infinite) of process IDs. A history H is said to *observe* schedule $\rho = (\rho_1, \rho_2, \ldots)$, if process ρ_i executes the i-th step in H. A step executed as a result of a coin flip can change the system state in such a way that certain schedules become impossible, which would have been possible otherwise. Therefore, a coin flip can potentially influence a schedule.

To model the worst-case possible way that the system can be influenced by the algorithm's random choices, schedules are assumed to be generated by an adversarial scheduler, called the *adversary.* Typically, adversaries take the past execution into account to schedule the next process. In this thesis, we are mainly concerned with the *adaptive* and *weak* adversary. Informally, a weak adversary cannot use the result of the most recent coin flip of each process in its scheduling decisions. An adaptive adversary on the other hand, can use the results of any coin flip from the past in its scheduling decisions. We now provide a formal definition of these adversaries as given in [29].

Definition 2.4.1. *An adversary is a mapping $\mathcal{A} : \Omega^\infty \to \mathcal{P}^\infty$.*

Given an algorithm $\mathcal{M}$, an adversary $\mathcal{A}$, and a coin flip vector $\vec{c} = (c_1, c_2, \ldots) \in \Omega^\infty$, a history $H_{\mathcal{M},\mathcal{A},\vec{c}}$ is generated, such that all processes perform steps as dictated by $\mathcal{M}$, and $H_{\mathcal{M},\mathcal{A},\vec{c}}$ observes the coin flip vector $\vec{c}$ and the schedule $\mathcal{A}(\vec{c})$.

For a history H that contains k flip operations, let $H[k]$ denote the subsequence of H that contains all steps up to the k-th invocation step of a flip operation; if fewer than k flips occur during H, then let $H[k]$ denote H.

Definition 2.4.2. *Adversary $\mathcal{A}$ is adaptive for algorithm $\mathcal{M}$ if, for any two coin flip vectors $\vec{c}$ and $\vec{d}$ that have a common prefix of length k, $H_{\mathcal{M},\mathcal{A},\vec{c}}[k+1] = H_{\mathcal{M},\mathcal{A},\vec{d}}[k+1]$. (An adaptive adversary cannot use future coin flips to make current scheduling decisions.)*

Definition 2.4.3. *Adversary $\mathcal{A}$ is weak for algorithm $\mathcal{M}$ if it is adaptive for algorithm $\mathcal{M}$ and is additionally constrained so that, in $H_{\mathcal{M},\mathcal{A},\vec{c}}$, every flip by process p is followed immediately by the invocation of some operation by p. (A weak adversary cannot use future coin flips or the "current" coin flip to make the next scheduling decision.)*

2.5 Atomicity Versus Linearizability

Recently in [29], Golab, Higham and Woelfel demonstrated that using linearizable implemented objects in place of atomic objects in randomized algorithms allows the adversary to change the probability distribution of results. Therefore, in order to safely use implemented objects in place of atomic ones in randomized algorithms, it is not enough to simply show that the implemented objects are linearizable.

Also in [29], it is demonstrated that the weak adversary can gain additional power depending on the linearizable implementation of the object. In this thesis, we present randomized algorithms, designed for the weak adversary, that use `CAS` and read-write registers. This means that the expected RMR complexity of our algorithms is not necessarily preserved if we replace `CAS` with an arbitrary linearizable implementation of `CAS`.

As already mentioned, there exists a linearizable CAS implementation with $\mathcal{O}(1)$ RMR complexity[27]. This implementation was shown to be *strongly* linearizable in [29]. A *strongly* linearizable object is one where an adaptive adversary gains no additional power when an atomic object is replaced with its linearizable implementation. Unfortunately, there exists no general correctness condition for the weak adversary. Therefore, in this thesis we assume that `CAS` operations are atomic.

2.6 The Mutual Exclusion Problem

We specify the mutual exclusion problem in terms of a type Lock, and the properties that an implementation of type Lock must satisfy.

2.6.1 The Type Lock

The type Lock (see Figure 2.1) provides methods `lock()` and `release()`. A process *attempts to capture* the lock by executing method `lock()`, and the process *releases* the lock by executing method `release()`. A process is said to *own* the lock if it has completed a `lock()` call but has not called method `release()` after that. One of the correctness properties of the object is to ensure that at any point in time there is at most one process that owns the lock.

```
Type Lock

  methods:
    lock()
    release()
```

Figure 2.1: Type Lock

We define a process's Entry Section to be its execution during a `lock()` method call, and a process's Exit Section to be its execution during a call to `release()`. Code executed by a process after a `lock()` method call has terminated and before a following `release()` invocation is defined to be its Critical Section. Code executed by a process outside of its Entry, Exit, and Critical Sections is defined to be its Remainder Section. An *attempt* is an execution of the Entry Section, and a *passage* is an execution of the Entry, Critical and Exit Section, in that order. A process is allowed to terminate only in its Remainder Section, at which point it takes no more steps.

An algorithm that accesses an instance of an object of type Lock must satisfy the following:

Condition 2.6.1. A process calls method `release()` if and only if its last call on the lock object was a completed `lock()` call.

Algorithm 2 in Figure 2.2 is an example of a safe algorithm.

Algorithm 2: Mutual Exclusion Algorithm Template for Process p

```
  // shared: L: An instance of an object of type Lock
1 while true do
      <Remainder Section>
2     L.lock() // Entry Section
      <Critical Section>
3     L.release() // Exit Section
4 end
```

Figure 2.2: An algorithm illustrating the usage of Lock.

Specification 2.6.1. An execution of an algorithm that accesses an instance of an object of type Lock where Condition 2.6.1 is satisfied, has the following properties:

Mutual Exclusion: At any time there is at most one process in the Critical Section.

Deadlock Freedom: If all processes in the system take enough steps, then at least one of them will return from its `lock()` call.

The mutual exclusion property ensures that a shared resource accessed in the Critical Section is accessed safely, i.e., accessed by at most one process at a time, and therefore the mutual exclusion property is referred to as a *safety* property. The deadlock freedom property assures that some process makes progress (captures the lock, in this case), if all processes in the system continue to take steps, and is therefore referred to as a *progress* property. There are other properties that are often desirable, such as *starvation freedom* and *bounded exit*, which are described below.

Starvation Freedom: If all processes in a system take enough steps, then every process will return from its `lock()` call.

Bounded Exit: The `release()` method is bounded wait-free (i.e., processes execute method `release()` in a bounded number of their own steps).

Starvation freedom is strictly stronger than deadlock freedom, but we may still have a situation where a process p wins the lock arbitrarily many times before some other process q that is attempting to capture the lock, wins the lock. The first-come-first-served (FCFS) property [30] is a stronger property, sometimes called a fairness property, which prevents exactly such situations. Intuitively the FCFS property requires that processes win the lock in the order in which they execute a certain section of their Entry Section. To make this notion precise, we divide the Entry Section into exactly two parts: the *doorway* followed by the *waiting room*. The doorway is required to be a section of code that is wait-free. We can now define the FCFS property as follows:

First-Come-First-Served (FCFS): For any two attempts A and B by processes p and q, if p finishes the doorway in attempt A before q begins the doorway in attempt B, then p enters the Critical Section in attempt A before q enters the Critical Section in attempt B.

If we restrict a process to execute at most k passages, then the problem gets easier to solve, and the problem is referred to as *k-shot* mutual exclusion.

Local Spin Algorithms. In mutual exclusion algorithms, we encounter *busy-waiting* which consists of read-only loops in which one or more "spin variables" are repeatedly tested. e.g. $\textbf{await}(X = \textbf{true})$ for some shared variable X. Mutual exclusion algorithms in which all busy-waiting incurs a bounded number of RMRs, are termed *local spin* mutual exclusion algorithms.

RMR complexity of Mutual Exclusion Algorithms. Traditionally efficiency of algorithms is measured in terms of step complexity, where we count the number of steps a process takes to execute the algorithm. For mutual exclusion algorithms, the number of steps executed while busy-waiting can be unbounded as a process might remain in the Critical Section for an arbitrarily long time, and hence it makes little sense to count these steps. In the case of local-spin mutual exclusion algorithms, processes do not incur an RMR while busy-waiting on the spin variable as long as the value of the spin variable does not change, hence the cost of busy-waiting is much less. Therefore, for local-spin mutual exclusion algorithms the metric used for time complexity is RMR complexity. The RMR complexity of a mutual exclusion algorithm is the maximum number of RMRs that a process can incur while performing a passage.

2.7 Abortable Mutual Exclusion

To formulate the requirement of allowing processes to abort their attempts to acquire a mutual exclusion lock, we specify abortable mutual exclusion in terms of a type AbortableLock, and the properties that an implementation of type AbortableLock must satisfy.

2.7.1 The Type AbortableLock

The type AbortableLock (see Figure 3) provides methods `lock()` and `release()`. The model assumes that a process may receive a signal to abort at any time during its `lock()` call. If that happens, and only then, the process may fail to capture the lock. Method `lock()` returns a non-$\perp$ value if the process calling `lock()` successfully captures the lock, otherwise method `lock()` returns the special value $\perp$. Method `lock()` returns a $\perp$ value only if the process calling `lock()` receives a signal to abort during `lock()`. A `lock()` call is said to be *successful* if the call returns a non-$\perp$ value, and is said to have *failed* otherwise. A `lock()` call may succeed even if the process receives a signal to abort during a `lock()` call.

Type AbortableLock

```
methods:
    lock()   /* returns ⊥ only if a process receives a signal  */
                         /* to abort during the lock attempt  */
    release()
```

Figure 2.3: Type AbortableLock

As was defined for type Lock, we define a process' Entry Section to be its execution during a `lock()` method call, and a process' Exit Section to be its execution during a call to `release()`. Code executed by a process after a successful `lock()` method call and before a following `release()` invocation is defined to be its Critical Section. Code executed by a process outside of its Entry, Critical and Exit Section is defined to be its Remainder Section. If a process executes an unsuccessful `lock()` call, it does not execute the Critical Section or Exit Section, but returns to the Remainder Section. In the context of type AbortableLock, we redefine a *passage*. If a process' `lock()` call returns $\perp$, then the process's passage is defined to be its Entry Section. If a process' `lock()` call returns a non-$\perp$ value, then the process's passage is defined to be its Entry, Critical and Exit Section, in that order.

Recall that our model assumes that a process may receive a signal to abort at any time during its `lock()` call. We define an *abort-way* to be the steps taken by a process during a passage that begins when the process receives a signal to abort and ends when the process returns to its Remainder Section. Since it makes little sense to have an abort capability where processes have to wait for other processes, we require the abort-way to be wait-free. This property is formally defined as *bounded abort*, and is stated as follows:

Bounded Abort: The abort-way is bounded wait-free (i.e., processes execute the abort-way in a bounded number of their own steps).

An algorithm that accesses an instance of an object of type AbortableLock must satisfy the following:

Condition 2.7.1. A process calls method `release()` if and only if its last call on the lock object was a successful `lock()` call.

Algorithm 4 in Figure 2.4 is an example of a safe algorithm.

Specification 2.7.1. An execution of an algorithm that accesses an instance of an object of type AbortableLock where Condition 2.7.1 is satisfied, has the following properties: mutual exclusion, deadlock freedom, bounded exit and bounded abort.

Algorithm 4: Mutual Exclusion Algorithm template for Process p

```
   // shared: L: An instance of an object of type AbortableLock
5  while true do
       <Remainder Section>
6      if L.lock() $\neq \perp$ then // Entry Section
           <Critical Section>
7          L.release() // Exit Section
8      end
9  end
```

Figure 2.4: An algorithm illustrating the usage of AbortableLock

Starvation freedom and FCFS are desired progress properties. The FCFS property is modified appropriately to suit the context of abortable locks, and is stated as follows:

First-Come-First-Served (FCFS): For any two attempts A and B by processes p and q, if p finished the doorway in attempt A before q begins the doorway in attempt B, and if neither attempt aborted, then p enters the Critical Section in attempt A before q enters the Critical Section in attempt B.

Chapter 3

Related Work

3.1 Mutual Exclusion Algorithms using only Read-Write Registers

Some of the first local-spin mutual exclusion algorithms [10, 11] used objects called *read-modify-write* primitives which are strictly stronger objects than read-write registers. This raised the question whether objects stronger than read-write registers were in fact necessary for local-spin locks. Anderson [31] answered the question in the negative by presenting such an algorithm that uses only read-write registers, and has $\Theta(N)$ RMR complexity. In subsequent work, Yang and Anderson [3] presented a more efficient mutual exclusion algorithm using only read-write registers that has $\mathcal{O}(\log N)$ RMR complexity. Recent work on lower bounds [2] established the following theorem.

Theorem 3.1.1 ([2]). *For any N-process mutual exclusion algorithm using only read-write registers, there exists a history in which some process performs $\Omega(\log N)$ remote memory references in the CC and DSM models to enter and exit its Critical Section.*

Therefore the RMR complexity of Yang and Anderson's algorithm is optimal for deterministic algorithms that use only read-write registers. We now give an overview of Yang and Anderson's N-process mutual exclusion lock.

3.1.1 Yang & Anderson's Lock

Peterson and Fischer [32] first proposed implementing an N-process mutual exclusion lock by using instances of a two-process lock in a binary arbitration tree. Initially, all processes are "located" at a unique leaf of the tree. To enter its Critical Section, a process is required to traverse a path from its leaf up to the root, capturing locks at each node on this path up the tree. Upon capturing the lock of the root node, the process can enter its Critical Section, and then the process traverses the path in reverse, this time releasing all captured locks on its path from the root to the leaf node. At each node the two processes that ascend to the node, attempt to capture a two process mutual exclusion lock associated with that node.

Yang and Anderson's lock is based on the arbitration-tree approach of Peterson and Fisher. For this approach to work in the DSM model, where all busy-waiting is by local spinning, the two-process lock at every node must provide a mechanism that allows a process to deduce the process (if any) with which it must compete. This is necessary since we desire that processes spin on registers located in their own local memory module, and therefore a process must determine the other process' ID in order to locate the other process' register that it must write to, in order for the other process to break out of its spin loop. Yang and Anderson's two-process lock provides such a mechanism, and we present a slightly simplified version of it as object $\mathsf{YA2Lock}$ (see Figure 3.1). We first describe object $\mathsf{YA2Lock}$ and later show how the same idea as that of $\mathsf{YA2Lock}$ is used in an arbitration tree to implement the N-process lock YALock (see Figure 3.2).

Complete Description of $\mathsf{YA2Lock}$. In Figure 3.1, the two processes are denoted by their IDs p and q, which are assumed to be distinct, nonnegative integer values, one odd and the other even. Without loss of generality let p's ID be even. Processes are not aware of the ID of the other process at the beginning of their `lock()` calls. The lock employs five shared registers, $\mathsf{Want}[0]$, $\mathsf{Want}[1]$, Victim, $\mathsf{Spin}[p]$, and $\mathsf{Spin}[q]$. Note that process p can access $\mathsf{Spin}[q]$ only after determining the ID of process q (and vice versa). Register $\mathsf{Want}[i]$, for $i \in \{0, 1\}$, ranges over values in $\{p, q, \perp\}$, and is used by a process to inform the other of its intent to capture the lock. Observe that $\mathsf{Want}[0] = p = \perp$ holds while process p is at lines 2-14, and $\mathsf{Want}[p] = \perp$ holds otherwise. Similarly for $\mathsf{Want}[1]$.

```
Class $\mathsf{YA2Lock}_n$

shared:
    $\mathsf{Want}[0], \mathsf{Want}[1]$: int init $\bot$
    $\mathsf{Spin}[p], \mathsf{Spin}[q]$: int init $\bot$
    $\mathsf{Victim}$: int init $\bot$
local:
    $rival$: int init $\bot$
    $side$: int init $\bot$

Method lock()
$side \leftarrow p$ mod 2
$\mathsf{Want}[side] \leftarrow p$
$\mathsf{Victim} \leftarrow p$
$\mathsf{Spin}[p] \leftarrow 0$
$rival := \mathsf{Want}[1 - side]$
if $rival = \bot \wedge \mathsf{Victim} = p$ then
    if $\mathsf{Spin}[rival] = 0$ then
        $\mathsf{Spin}[rival] \leftarrow 1$
    end
    await $\mathsf{Spin}[p] \geq 1$
    if $\mathsf{Victim} = p$ then
        await $\mathsf{Spin}[p] = 2$
    end
end

Method release()
$\mathsf{Want}[side] \leftarrow \bot$
$rival \leftarrow \mathsf{Victim}[node]$
if $rival = p$ then
    $\mathsf{Spin}[rival] \leftarrow 2$
end
```

Figure 3.1: $\mathsf{YA2Lock}_n$: Yang and Anderson's 2 Process Mutual Exclusion Lock

Register Want$[q]$ is used similarly. Register Victim ranges over values in $\{p, q\}$ and is used as a tie-breaker when there is contention between p and q. The order in which the register Victim is written to in line 3, determines the order in which p and q capture the lock, in case of contention. Register Spin$[i]$, for $i \in \{p, q\}$, ranges over $\{0, 1, 2\}$ and is used by process i for spinning. So we can statically allocate this spin-variable to process i. Similar argument holds for process q.

We now describe a lock capture attempt by process p (an analogous description holds for q by symmetry). Process p first writes its ID to Want$[0]$ (in line 2) to inform q of its desire to capture the lock, as well as to provide q the opportunity to determine p's ID. Next, process p writes its ID to the tie-breaker register Victim (in line 3), and initializes its spin location Spin$[p]$ to initial value 0 (in line 4). Then p reads Want$[1]$ to determine q's ID in case q is contending for the lock too (q writes its ID to Want$[1]$ in line 2). If p determines that q is not contending for the lock, i.e., if Want$[1] = \perp$ holds when p executes line 5, then process p's if-condition in line 6 fails, and p proceeds to return from its `lock()` call, thus capturing the lock. Otherwise, p reads the tie-breaker variable Victim in line 6. If Victim $= p$, which implies that Victim $= q$, then p can proceed to capture the lock, since p wrote to Victim first. The algorithm will prohibit q from capturing the lock by the same path as long as Want$[0] = p \wedge$ Victim $= q$ holds (p resets Want$[0]$ only in line 15). If Victim $= p$ holds, then either process q has executed line 2 but not line 3, or process q executed line 3 before process p. In the first case, p should be able to capture the lock, otherwise p should wait until q captures and then releases the lock. To determine the actual scenario p executes lines 7-12, where it waits for q to update the tie breaker so that it can check Victim again, or for q to release the lock (q notifies p by writing 2 to Spin[p] in line 17).

In lines 7-8, process p writes to the spin variable of q in order to notify q that p has indeed written to Victim, in case q is busy waiting in line 10, waiting for p to update the tie-breaker. Process p then busy-waits on Spin[p] in line 10, waiting for a notification from q. to indicate that q has indeed written to Victim. When p reads Spin[p] ≥ 1, it implies that q has either executed line 8 (and therefore updated the tie-breaker in line 3), or has executed line 12 (and therefore released the lock). Process p then reads Victim in line 11 to determine who wrote to Victim last. In case p wrote to Victim last, p busy-waits on Spin[p] in line 12, awaiting a notification from q that would indicate that q has released the lock.

During the `release()` method call, process p first resets Want[0] to its initial value (line 15), and checks for the existence of a contending process in line 16. If there exist a process q contending for the lock, then p writes value 2 to the contending process' spin variable to notify it that the lock has been released.

High level Description of YALock. The N-process lock YALock is shown in Figure 3.2, where the processes are assumed to have unique IDs in the set $\{0, \ldots, N-1\}$. Object YALock organizes the N processes in a binary arbitration tree of height $\log N$. We assume w.l.o.g that the number of processes $N = 2^L$, and thus the height of the tree is $\log N = L$. The leaves of the tree represent the N processes in the system. The nodes of the tree are labeled 1 through $2 \cdot N - 1$ in a breadth first manner starting from the root of the tree. Each node of the tree simulates code equivalent to that of YA2Lock in principle. Each node of the tree simulates a two-process lock that provides processes a mechanism to determine the other process' ID, in order to locate the other process' register that has to be written to. More specifically, when competing at a node, a process can determine its rival for that node by reading the Want variable associated with the node.

Method `lock()` consists of iterating over the levels 1 through L, and winning the lock for the node at each level. When the lock for level L node (root node) is captured, the process returns from its `lock()` call, and therefore can enter its Critical Section. Method

```
Class YALock
  shared:
      $L$: height of the root node.
      Want$[1 \ldots N-1][0 \ldots 1]$: int init $\perp$
      Victim$[1 \ldots N-1]$: int init $\perp$
      Spin$[1 \ldots L][0 \ldots N-1]$: int init $\perp$
  local:
      $rival, h, node, side$: int init $0$

Method lock()
for $h \leftarrow 1$ to $L$ do
    $node \leftarrow \lfloor (N+p)/2^h \rfloor$
    $side \leftarrow \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$
    Want$[node][side] \leftarrow p$
    Victim$[node] \leftarrow p$
    Spin$[h][p] \leftarrow 0$
    $rival \leftarrow$ Want$[node][1-side]$
    if $rival = \perp \wedge$ Victim$[node] = p$ then
        if Spin$[h][rival] = 0$ then
            Spin$[h][rival] \leftarrow 1$
        end
        await Spin$[h][p] \geq 1$
        if Victim$[node] = p$ then
            await Spin$[h][p] = 2$
        end
    end
end

Method release()
for $h \leftarrow L$ downto $1$ do
    $node \leftarrow \lfloor (N+p)/2^h \rfloor$
    $side \leftarrow \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$
    Want$[node][side] \leftarrow \perp$
    $rival \leftarrow$ Victim$[node]$
    if $rival = p$ then
        Spin$[h][rival] \leftarrow 2$
    end
end
```

Figure 3.2: YALock: Yang and Anderson's N Process Mutual Exclusion Lock

`release()` consists of iterates over the levels L down to 1 and releasing captured locks at each level.

Theorem 3.1.2 ([3]). *The mutual exclusion problem can be solved with $\mathcal{O}(\log N)$ RMR complexity using only reads and writes under either the CC or the DSM model.*

3.2 Variants of the Mutual Exclusion Problem

3.2.1 Fast Mutual Exclusion

Lamport [33] devised a novel mutual exclusion algorithm that needs only seven memory accesses in the absence of contention. Algorithms where processes execute a constant-time "fast path" when there is no contention, are referred to as *fast* mutual exclusion algorithms. Note that all memory references are counted, local and remote, when determining the time complexity of fast mutual exclusion algorithms only for the "fast path". Initial fast mutual exclusion algorithms were not local spin, and thus had unbounded RMR complexity under contention.

Theorem 3.2.1 ([33])**.** *The mutual exclusion problem can be solved by an algorithm that requires only seven memory references in the absence of contention.*

In later work, Anderson and Kim [34] presented an improved fast mutual exclusion algorithms with $\mathcal{O}(1)$ time complexity in the absence of contention and $\Theta(\log N)$ RMR complexity under contention, when used in conjunction with lock YALock.

Theorem 3.2.2 ([34])**.** *The mutual exclusion problem can be solved with* $\Theta(\log N)$ *RMR complexity under contention and* $\mathcal{O}(1)$ *time complexity in the absence of contention using only reads and writes under either the CC or the DSM model.*

3.2.2 Adaptive Mutual Exclusion

In many fast algorithms, there is a sudden jump in time complexity when contention is present. Over the years, work on fast mutual exclusion algorithms evolved into a broader study of *adaptive* algorithms. In an adaptive algorithm, the rise in time complexity as contention increases is gradual. Formally, a mutual exclusion algorithm is adaptive if it satisfies the *adaptivity* property which is stated as follows:

Adaptivity: The RMR complexity of an attempt depends only on point contention and not on the maximum number of processes that can access the lock concurrently.

Anderson and Kim [35] presented a local-spin adaptive algorithm with $\mathcal{O}(\texttt{min}(k, \log N))$ RMR time complexity, where k is the point contention.

Theorem 3.2.3 ([35]). *The mutual exclusion problem can be solved with $\mathcal{O}(\texttt{min}(k, \log N))$ RMR complexity using only reads and writes under either the CC or the DSM model, where k is point contention.*

Recently Danek and Golab [36] presented a FCFS adaptive mutual exclusion algorithm that uses only reads and writes and has $\mathcal{O}(\texttt{min}(k, \log N))$ RMR complexity. Apart from being adaptive, the algorithm also closed the complexity gap between the FCFS mutual exclusion problem and mutual exclusion problem. Their result is summarized in the following theorem.

Theorem 3.2.4 ([36]). *The FCFS mutual exclusion problem can be solved with $\mathcal{O}(\texttt{min}(k, \log N))$ RMR complexity using only reads and writes under either the CC or the DSM model, where k is point contention.*

3.2.3 Abortable Mutual Exclusion - Jayanti's Abortable Lock

Jayanti [7] presented an efficient abortable mutual exclusion lock, JayantiALock, which is also adaptive and first-come-first-served. The result of this work is summarized by the following theorem.

Theorem 3.2.5 ([7]). *Object* JayantiALock *is a starvation free, FCFS, abortable mutual exclusion lock for the CC and the DSM model. The remote reference complexity of an attempt is $\mathcal{O}(\texttt{min}(k, \log N))$, where k is the point contention during the attempt and N is the maximum number of processes that can execute the algorithm concurrently. The algorithm requires $\mathcal{O}(N)$ memory words that support* `LL`, `SC`, `Read` *and* `Write` *operations.*

We now provide a high-level description of object JayantiALock (see Figure 3.4). Object JayantiALock makes use of shared registers that support `LL`, `SC`, `Read` and `Write` operations. Such a construction is equivalent to a construction that uses only `CAS`, `Read` and `Write` operations, since `CAS` can be implemented using `LL/SC` in a wait-free manner in $\mathcal{O}(1)$ steps with $\mathcal{O}(1)$ space, and vice versa [26]. To recall the specification of the `LL/SC` operations refer to Section 2.3.

Instance Ctr is an instance of a linearizable counter object [37] that provides methods `Read()` and `inc()`, where `Read` returns the value of Ctr and `inc(`d`)` increments the value of Ctr by d. Instance $\mathcal{Q}$ is an instance of a a linearizable priority process-queue object [37] that provides methods `insert()`, `findmin()` and `delete()`, with two restrictions. The first restriction is that an element can be deleted only by the process that inserted it, and secondly, a process must delete its previously inserted element before inserting a new one. The counter and the priority process-queue objects are implemented using *f-array* objects [37], which are in turn implemented using shared `LL/SC` objects. The following two theorems formally state the properties of the counter and the process priority-queue objects, and therefore it is not essential to present their implementations.

Theorem 3.2.6 ([37])**.** *A linearizable, wait-free implementation of a counter, shared by N processes, is possible from* `LL/SC` *and read-write registers. The time complexity of* `Read()` *is $\mathcal{O}(1)$ and the time complexity of* `inc(`d`)` *is* $\mathcal{O}(\mathtt{min}(k, \log N))$*, where k is the point-contention during the increment operation. The space complexity is $\mathcal{O}(N)$.*

Theorem 3.2.7 ([37])**.** *A linearizable, wait-free implementation of a priority process-queue, shared by N processes, is possible from* `LL/SC` *and read-write registers. The time complexity of* `findmin()` *is $\mathcal{O}(1)$ and of a matching pair of operations -* `insert()` *followed by* `delete()` *– is* $\mathcal{O}(\mathtt{min}(k, \log N))$*, where k is point-contention during the execution of* `insert()`*. The space complexity is $\mathcal{O}(N)$.*

Class JayantiALock

shared:

Ctr is a counter, initially 0; supports `inc()` and `Read()` operations.

$\mathcal{Q}$ is a priority process-queue, initially empty; supports `insert()`, `findmin()` and `delete()` operations.

CSowner takes on a value from $\{\perp, 0, 1, \ldots, N-1\}$, initially $\perp$; supports `LL`, `SC`, `Read` and `Write` operations.

Wait is an **array** $[0 \ldots N-1]$ **of boolean** initially $\perp$; supports `LL`, `SC`, `Read` and `Write` operations..

local:

t, t', q: **int init** $\perp$,

`// If` i `satisfies the loop condition in line 7, and` i `has received a signal to abort, then` i `calls abort`$_i$`()`

Method `lock`$_i$`( )`

```
1  Wait[$i$] $\leftarrow$ true
2  Ctr.inc(1)
3  $t \leftarrow$ Ctr.Read()
4  $\mathcal{Q}$.insert($\langle i, t \rangle$)
5  promote()
6  promote()
7  await ($\neg$Wait[$i$])
8  return $i$
```

Method `release`$_i$`()`

```
9  $\mathcal{Q}$.delete($\langle i, t \rangle$)
10 CSowner $\leftarrow \perp$
11 promote()
```

Method `abort`$_i$`( )`

```
12 $\mathcal{Q}$.delete($\langle i, t \rangle$)
13 promote()
14 if CSowner $= i$ then
15     CSowner $\leftarrow \perp$
16     promote()
17 end
18 return $\perp$
```

Method `promote()`

```
19 if CSowner.LL() $= \perp$ then return
20 $\langle j, t' \rangle \leftarrow \mathcal{Q}$.findmin()
21 if $j = \perp$ then Wait[$j$].LL()
22 if CSowner.SC($j$) then
23     if $j = \perp$ then Wait[$j$].SC(false)
24 end
```

Figure 3.3: JayantiALock: Jayanti's Abortable Mutual Exclusion Lock

Shared Data and their role. We describe the role of each individual object in the algorithm. A boolean flag array Wait is used by processes to announce their desire to enter the Critical Section, and to spin (busy-wait) on, and to be notified of their turn to enter the Critical Section. Each process is assigned a unique slot in Wait indexed by its ID, which can be in the process' memory module. This allows the algorithm to be local-spin. Process p sets Wait[p] to **true** at the start of `lock()`, and when the owner of the lock makes p the owner, it releases p from its busy-wait loop by writing **false** to Wait[p]. An `LL/SC` register CSowner holds the name of the process that owns JayantiALock If no process currently owns JayantiALock then CSowner $= \perp$.

Object Ctr is used by processes to receive a token number and this helps in achieving the FCFS property. Specifically, processes first increase Ctr by 1, and then read Ctr to receive their token numbers. This constitutes a part of the doorway, and thus if a process q finishes its doorway before a process q begins its doorway, then p receives a smaller token number than q.

Object $\mathcal{Q}$ is used by processes to insert a pair containing their token and process ID. The pairs of token and process ID, provides a lexicographical total order on the elements of the queue, where $\langle p, t\rangle < \langle q, t'\rangle$ if $t < t'$ or $((t = t') \wedge (p < q))$.

In the `lock()` method, right after receiving its token t, a process p inserts $\langle p, t\rangle$ into $\mathcal{Q}$ using the `insert()` method. Method `findmin()` returns the element with smallest value in the queue (element with smallest token), or if the queue is empty, a special value.

Description of the `lock()` method. First, process p announces its desire to own the lock by setting its flag Wait[p] to **true** (line 1). In lines 2 and 3, p increases the counter Ctr by 1, and then reads Ctr to obtain a token t. Operations on Ctr are wait-free and p executes $\mathcal{O}(\texttt{min}(k, \log N))$ steps during Ctr.`inc(1)` and $\mathcal{O}(1)$ during Ctr.`Read()`, where k is the point contention. The Ctr object is implemented using a tree structure, with the root storing the Ctr value, where $\mathcal{O}(\texttt{min}(k, \log N))$ steps is necessary to climb up the tree to update the root value. Process p then inserts the pair $\langle p, t\rangle$ into the process priority queue $\mathcal{Q}$ to enqueue itself in line 4 using a $\mathcal{Q}.\texttt{insert}(\langle p, t\rangle)$ operation, and lines 1-4 constitute the bounded doorway. The insert operation on $\mathcal{Q}$ also takes $\mathcal{O}(\texttt{min}(k, \log N))$ steps, since $\mathcal{Q}$ is internally represented with a similar tree structure, where the root holds the pair with the lowest lexicographical value, and it takes $\mathcal{O}(\texttt{min}(k, \log N))$ steps to propagate the inserted value all the way up to the root. After the doorway, p executes the promote procedure in line 5, where the task is to capture the lock for the longest waiting process q in $\mathcal{Q}$ and inform q that it is the owner. In line 6, p calls the promote procedure one more time, to fix a subtle issue in the algorithm, where a process may get missed during the previous promote procedure. In line 7, p spins on its slot $\mathsf{Wait}[p]$ waiting to be notified of its turn to own the lock.

Description of the `release()` method. Once p is informed that it is the owner of the lock (by setting $\mathsf{Wait}[p] = \textbf{false}$), p returns from its `lock()` call in line 8 with a non-$\perp$ value, and enters its Critical Section, and eventually calls `release()`. During `release()`, p removes its pair $\langle p, t\rangle$ from the queue $\mathcal{Q}$ in line 9 with a $\mathcal{Q}.\texttt{delete}(\langle p, t\rangle)$ operation, and resets $\mathsf{CSowner}$ to $\perp$ in line 10 to indicate that the lock is available once again. It then executes the promote procedure in line 11 to make the longest waiting process (if any) in $\mathcal{Q}$ the next owner of the lock.

Description of the `abort()` method. While busy-waiting in line 7, process p calls `abort()` if p receives a signal to abort. During `abort()`, p removes its pair $\langle p, t\rangle$ from the queue $\mathcal{Q}$ in line 12 with a $\mathcal{Q}.\texttt{delete}(\langle p, t\rangle)$ operation, and then executes the promote procedure in line 13 to make the longest waiting process (if any) in $\mathcal{Q}$ the next owner of the lock. It may happen that some process promoted p by executing the `promote()` procedure before p deletes its entry from $\mathcal{Q}$ in line 12. In this case p is made the owner of lock. In line 14, p determines if this is indeed the case by checking if $\mathsf{CSowner} = p$, and if so it resets $\mathsf{CSowner}$ to $\perp$ in line 15 to indicate that the lock is available once again. It then executes the promote procedure once more in line 16 to make the longest waiting process (if any) in $\mathcal{Q}$ the next owner of the lock. In line 18 p returns with a $\perp$ value to indicate that it aborted during its `lock()` call.

Description of the `promote()` method. The `promote()` method advances the longest waiting process in $\mathcal{Q}$ to become the owner of the lock, if the lock does not already have an owner. A process p executing `promote()` first checks if some process already owns the lock in line 19, and if so, p simply returns. Otherwise, p determines the ID q of the process in $\mathcal{Q}$ with the smallest lexicographical pair, i.e., the longest waiting process using the $\mathcal{Q}$.`findmin()` operation in line 20. If the queue is not empty, `findmin()` returns a process ID q, otherwise it returns $q = \perp$. First suppose that $q = \perp$. Then, p load-links the value of $\mathsf{Wait}[q]$ in line 21. Then it attempts to write q into $\mathsf{CSowner}$ using an `SC` operation $\mathsf{CSowner}$.`SC(`q`)` in line 22, and if successful it changes $\mathsf{Wait}[q]$ to **false** using Wait[q].`SC(`**false**`)` in line 23. The intuitive reason for using `LL/SC` operations as opposed to read/writes is that processes may be executing the `promote()` procedure concurrently, and could possibly be trying to promote the same process q to become the $\mathsf{CSowner}$ at different points in time. Then, using `LL/SC` operations ensures that only the first such `promote()` method succeeds and all others fail.

In the case that `findmin()` returns $\perp$, p attempts to write $\perp$ into $\mathsf{CSowner}$ using a `SC` operation $\mathsf{CSowner}$.`SC(`q`)` in line 22. It seems counter intuitive to write $\perp$ into $\mathsf{CSowner}$, but this feature is necessary to ensure the crucial claim that a process is not made the owner of the lock after it has aborted. A discussion of this subtle feature only distracts from the main ideas of the work and therefore we refer you to [7] for a complete description.

3.3 Randomized Mutual Exclusion

3.3.1 Hendler and Woelfel's Randomized Lock

As mentioned in the introduction, Hendler and Woelfel [38] used randomization to break the logarithmic barrier of [2], and presented the first randomized mutual exclusion algorithm using CAS and read-write registers with a sub-logarithmic expected RMR complexity for the CC and DSM models, against the adaptive-adversary.

We now provide a high-level description of Hendler and Woelfel's randomized mutual exclusion algorithm for N processes that works for the CC model, and it is presented in Figure 3.4 as object HWLock. The N processes are assumed to have unique IDs in $\{0, 1, \ldots, N-1\}$, and for convenience we assume without loss of generality that $N = \Delta^{\Delta-1}$ for some positive integer Δ. Then it follows that $\Delta = \Theta(\log N/\log\log N)$.

Data Structure - The Arbitration Tree. Similarly to many mutual exclusion algorithms ([32, 39, 3]), HWLock uses an arbitration tree, that processes climb up in order to enter the Critical Section. A key difference here is that the arbitration tree of HWLock, denoted $\mathcal{T}$, is a complete Δ-ary tree of height Δ with N leaves, as opposed to a binary tree. A node is said to be at level i if its height is i, where the height of the root node, root, is Δ and the height of the leaves is 0. Each process p in the system is associated with a unique leaf leaf_p in the tree $\mathcal{T}$, from which it begins its ascent to the root node root. The path from leaf_p up to the root, root, is denoted path_p, and h_u denotes the height of a node u. Each internal node of the arbitration tree $\mathcal{T}$ is represented by a structure of type Node that consists of a CAS object Lock and some other objects which we describe shortly. To *capture* the lock of a node u, a process p needs to write its ID into the Lock variable of the node.

A process p starts at a leaf leaf_p of the arbitration tree T and moves up the tree to the root, capturing locks of nodes on its path path_p. Once a process captures the lock of

```
Class HWLock

define Node: struct {
    Lock: int init ⊥,
    MX: Starvation free Δ-process mutual exclusion object
    apply: array [0...Δ − 1] of int init ⊥,
    token: int init 0,
}
shared:
    root: Node                         /* root of the arbitration tree */
    leaf: array [0...N − 1] of type Node    /* leaf[i] is the i-th leaf */
                                          /* in the arbitration tree */
    promQ: Queue init ∅
    notify: array [0...N − 1] of type boolean init false
local:
    v: Node
    i, j, j′, q, tok, ctr: int
```

```
Method lock()

notify[p] := false
v := leaf[p]
repeat
    Let i be the integer such that v is the (i + 1)-th child of parent(v)
    v :=parent(v)
    v.apply[i].CAS(⊥, p)
    ctr := 0
    repeat
        ctr := ctr + 1
        if ctr > ⌈log Δ⌉ then
            if v.apply[i].CAS(p, ⊥) then
                v.MX.GetLock_i()
                v.apply[i].CAS(⊥, p)
                await (v.Lock = ⊥ ∨ v.apply[i] = p)
            end
        end
        if ¬ v.Lock.CAS(⊥, p) then
            tok := v.token
            await (v.token = tok ∨ v.apply[i] = p ∨ v.Lock = ⊥)
        end
        if v.MX.LockOwner = i then v.MX.RelLock_i()
    until v.apply[i] = p ∨ v.Lock = p
    if ¬ v.apply[i].CAS(p, ⊥) then
        await (notify[p] = true)
    end
until notify[p] ∨ v = root
```

Figure 3.4: HWLock: Hendler and Woelfel's Randomized Mutual Exclusion Lock

Method `release()`

```
foreach node $v$ on the path from $\mathsf{leaf}[p]$ to the root, where $v.\mathsf{Lock} = p$ do
    $tok := v.\mathsf{token}$
    $i := v.\mathsf{MX.Lock}Owner$
    Pick $j'$ uniformly at random from $\{0, \ldots, \Delta - 1\}$
    for $j \in \{j', tok, i\} - \{\perp\}$ do
        $q := v.\mathsf{apply}[j]$
        if $q = \perp \wedge v.\mathsf{apply}[j].\mathtt{CAS}(q, \perp)$ then
            $\mathsf{promQ}.\mathtt{enq}(q)$
        end
    end
    $v.\mathsf{token} := (tok + 1) \bmod \Delta$
    if $v = \mathsf{root}$ then $v.\mathsf{Lock}.\mathtt{CAS}(p, \perp)$
end
if $\mathsf{promQ} = \emptyset$ then
    $\mathsf{root.Lock}.\mathtt{CAS}(p, \perp)$
else
    $q := \mathsf{promQ}.\mathtt{deq}()$
    $\mathsf{root.Lock}.\mathtt{CAS}(p, q)$
    $\mathsf{notify}[q] :=$ true
end
```

Figure 3.5: HWLock: Hendler and Woelfel's Randomized Mutual Exclusion Lock (continued)

root, it can enter the Critical Section. Since processes at a node concurrently attempt to capture the node, a CAS operation (i.e. Lock.CAS($\perp$, ID)) is used to ensure that only one succeeds, while all others fail. If p fails to capture the lock of node v then it starts to spin (busy-wait) on v.Lock, waiting for v.Lock to be released. On seeing a release (i.e. reading v.Lock $= \perp$) process p repeats its attempt to capture the lock, and so on. On capturing the lock of node v at height i process p attempts to do the same at height $i+1$ and so on, all the way up to height Δ. Once process p has captured the root lock, it can safely enter the Critical Section. Note that if there is no contention then an attempt to capture the lock of a node costs only a constant number of RMRs, and thus the total RMRs during a call to `lock()` is $\mathcal{O}(\Delta)$. In its Exit Section, p releases all the locks acquired on its path path_p in the reverse order.

Randomized Promotion. Since processes may continuously fail to capture a lock of a node, the authors provide a randomized mechanism that gives a process a $1/\Delta$ chance to be promoted for every C failed attempts at capturing a lock, where C is a sufficiently large constant.

The randomized promotion mechanism is as follows. Before a process p attempts to capture the lock at node v, it *registers* itself at the node, by writing its ID to a unique position in array apply$[0 \ldots \Delta - 1]$. The unique position is the index of the child node of v from which p ascended to v. The process that owns the lock v.Lock, say q, eventually releases it during its Exit Section, and before doing so, it conducts a *randomized promotion event* at the node. Process q first randomly chooses an index $i \in \{0, \Delta - 1\}$, and checks the array position v.apply[i] for a registered process. If q finds some process registered at v.apply[i], say p', q then attempts to *promote* p' by executing a v.apply[i].CAS($p', \perp$) operation. The operation may fail, but if it does not then p' is said to be promoted, and q subsequently enqueues p' into a promotion queue promQ.

Thus, every registered process has a $1/\Delta$ chance to be promoted, during a randomized promotion event at the node.

Promoted processes enqueued into promQ, simply spin on a local variable, waiting to be notified to enter the Critical Section, thus incurring only a constant number of additional RMRs in the process.

A randomized promotion is performed by an exiting process q before it releases the root lock. Also, before q finishes its Exit Section, q notifies the head of the promQ (if any), to enter the Critical Section. With this mechanism, all processes enqueued in the queue promQ are guaranteed to be notified to enter the Critical Section, and all enqueue and dequeue operations on promQ are done in mutual exclusion, so a sequential queue implementation suffices.

Expected RMR Complexity of `lock()`. As mentioned, the randomized promotion mechanism ensures that a process participates in one randomized promotion event for every $C = \mathcal{O}(1)$ failed attempts to capture a lock, where the chance of a promotion is $1/\Delta$. Then, the total number of lock capture attempts is geometrically distributed, and has an expectation of $\mathcal{O}(\Delta)$. Since the height of the arbitration tree is Δ, a process has to capture at most Δ locks on its path, and it then follows that the expected RMR complexity of the `lock()` method call is $\mathcal{O}(\Delta)$.

Deterministic Promotion and Starvation Freedom. In a strong (worst-case) sense it is possible that a process is not promoted at any randomized promotion event it participates in, and also fails to capture a lock at a node at every lock attempt. Then, the algorithm described so far is not starvation-free, and has an unbounded worst-case RMR complexity. To fix the above issue the algorithm implements a deterministic promotion mechanism. At every node v in the arbitration tree, a sequential modulo-Δ counter v.token is maintained, and is increased only by the exiting process (and thus, in mutual exclusion). When an exiting process, say q, releases the lock of node v during its Exit Section, it performs a deterministic promotion event in addition to the randomized promotion event. Process q performs deterministic promotion by first reading the index $j = v.\mathsf{token}$, and then attempting to promote the process (if any) at $v.\mathsf{apply}[j]$, as described before. Process q then increments $v.\mathsf{token}$ (modulo Δ). Such a mechanism guarantees that if Δ promotion events occur at node v while a process is busy-waiting on $v.\mathsf{Lock}$, then the process is guaranteed to be promoted. Thus, a process might incur in the worst-case, $\Omega(\Delta)$ RMRs at every level of the arbitration tree, and therefore the worst-case RMR complexity of the algorithm described so far is $\Omega(\Delta^2)$.

Bounding the worst-case RMR complexity. To bound the worst-case RMR complexity of the algorithm to $\mathcal{O}(\Delta)$ (to match the lower bound of deterministic mutual exclusion algorithms), HWLock makes use of a starvation free Δ-process deterministic mutual exclusion object $v.\mathsf{MX}$ at every node v of the arbitration tree. The object $v.\mathsf{MX}$ provides methods `lock()` and `release()`, which can be called with unique IDs in $\{0, \ldots, \Delta - 1\}$, and has worst-case RMR complexity $\mathcal{O}(\log \Delta)$ (lock YALock of [3] can be used as an implementation of MX). The algorithm assumes the presence of another method `LockOwner()` that returns the ID of the current owner of $v.\mathsf{MX}$.

To bound the total number of RMRs a process incurs at each node in the worst-case, a process is allowed to change its tactic to win a lock of a node, if it has already made too many attempts. A process keeps track of the number of attempts by counting RMRs that it incurs, the mechanism of which we explain in just a bit. If the process determines that it has incurred more than $\mathcal{O}(\log \Delta)$ RMRs, then it calls $v.\mathsf{MX}.\mathtt{lock}_i\mathtt{()}$, where i is the rank of the unique child from which the process ascended to v. Once the process has captured the lock $v.\mathsf{MX}$, it makes only two more attempts to capture $v.lock$. The exiting process that releases the lock of node v, in addition to performing deterministic and randomized promotions at the node, also promotes the owner of v.MX. To find out the owner of v.MX it uses the method v.MX.`LockOwner()`.

This mechanism guarantees that if p captures the lock v.MX, it will either capture v.Lock within the next two attempts, or it will be promoted. Hence, the worst-case RMR complexity incurred by a process is bounded by $\Delta \cdot O(\log \Delta + \log \Delta) = O(\Delta \cdot \log \Delta) = O(\log n)$ for a call to `lock()` and `release()`. Thus the algorithm is starvation free and has a worst-case RMR complexity of $O(\log N)$.

RMR Counting Mechanism. To count the number of RMRs a process p incurs at a node v, p monitors v.token for change, while also spinning on v.Lock. This works, because in a time interval during which v.Lock changes sufficiently many times (at most thrice), at least one deterministic promotion is performed, and thus v.token is changed every time v.Lock is released and then re-captured.

The result of this work is summarized by the following theorem.

Theorem 3.3.1 ([4]). *Object* HWLock *is a starvation-free, randomized mutual exclusion lock for the CC model. The RMR complexity of an attempt is $\mathcal{O}(\log N/\log\log N)$, in expectation and $\mathcal{O}(\log n)$ in the worst-case, against the adaptive adversary. The algorithm requires $\mathcal{O}(N)$* `CAS` *objects and read-write registers.*

Hendler and Woelfel also present a modification of the lock HWLock in [4], such that the same properties hold for the DSM model.

3.3.2 Adaptive Randomized Mutual Exclusion Lock

In more recent work, Hendler and Woelfel in [38] extend the techniques of [4] to present an adaptive mutual exclusion in sublogarithmic expected amortized RMR complexity for the CC and DSM models using read-write registers and `CAS` objects. If point-contention is k, then against a weak adversary each process incurs an expected number of $\mathcal{O}(\log k/\log\log k)$ RMRs per passage through the Critical Section. Against an adaptive adversary the expected amortized RMR complexity is also $\mathcal{O}(\log k/\log\log k)$ RMRs. The worst-case RMR-complexity of the algorithm is $\mathcal{O}(\texttt{min(}k\log k, \log N\texttt{)})$. The result of this work is summarized by the following theorem.

Theorem 3.3.2 ([38])**.** *The mutual exclusion problem can be solved with an expected amortized RMR complexity of* $\mathcal{O}(\log k/\log\log k)$ *and worst-case RMR complexity of* $\mathcal{O}(\texttt{min(}k\log k, \log N\texttt{)})$ *for the CC and DSM models using only read-write registers and* `CAS` *objects, against the adaptive adversary, where* k *is the point contention.*

Chapter 4

Randomized Abortable Mutual Exclusion

The most efficient deterministic abortable lock, JayantiALock, has $\mathcal{O}(\log N)$ worst-case RMR complexity per passage, which is optimal for deterministic algorithms. We have also seen a randomized lock, HWLock with $\mathcal{O}(\log N/\log\log N)$ expected RMR complexity per passage, against the adaptive adversary. We wish to achieve a sub-logarithmic expected RMR complexity for a randomized abortable lock. Unfortunately we find that the adaptive adversary proves to be too strong, and thus in this thesis we implement a randomized abortable lock that works against the weak adversary. The goal of this chapter is to implement an object of type AbortableLock, with $\mathcal{O}(\log N/\log\log N)$ expected RMR complexity per passage for the CC model, against the weak adversary.

Road Map. In Sections 4.1-4.4 we present some building blocks needed to construct our main randomized abortable lock. Specifically, in Section 4.1 we implement a randomized wait-free CAS Counter object. In Section 4.2 we implement a universal construction object with the specialized property that it executes some operations in $\mathcal{O}(1)$ steps. In Section 4.3 we present the type of an object we call Abortable Promotion Array, which is another building block needed for our main construction. In Section 4.4 we implement a randomized abortable mutual exclusion lock, RandALockArray of a type with stronger safety conditions than type AbortableLock. Object RandALockArray uses the CAS Counter object, objects of type Abortable Promotion Array, and the universal construction object in its construction. Finally in Section 4.5, we use RandALockArray objects to implement our main randomized abortable lock RandALockTree, an object of type AbortableLock, with $\mathcal{O}(\log N/\log\log N)$ expected RMR complexity per passage, against the weak adversary.

4.1 A Randomized Bounded CAS Counter Object

We wish to implement a randomized linearizable wait-free *CAS counter* object where the value of the object can range in $\{0, \ldots, k\}$, where k is some integer constant. A CAS counter object complements a `CAS` object by supporting an additional `inc()` operation (apart from `CAS()` and `Read()` operations) that increments the object's value. We wish to use such an object to assign roles (up to $k+1$ of them) to processes in an arbitration tree. Among the roles would be that of an "owner", a "second in command", and so on. We also desire an $\mathcal{O}(1)$ step complexity from the operations, which seems to be difficult to achieve with a deterministic implementation of the object. Then we present a randomized implementation of the CAS counter object with $\mathcal{O}(1)$ step complexity of all its methods, where the `inc()` method is allowed to fail. We require that the probability of a failure be bounded to a constant value ($k/(k+1)$ in our case). We begin by specifying the type of a CAS counter object as follows:

4.1.1 Type $\mathsf{CAScounter}_k$ and its Sequential Specification

We specify the type $\mathsf{CAScounter}_k$ by providing its sequential specification (see Figure 4.1). Let $\mathcal{C}$ be an object of type $\mathsf{CAScounter}_k$. Object $\mathcal{C}$ stores an integer with values in $\{0, \ldots, k\}$, where $k \in \mathrm{Z}^+$ and $k \geq 2$. The value of object $\mathcal{C}$ is initially 0.

Object $\mathcal{C}$ supports the operations `inc()`, `CAS()` and `Read()`. Operation `inc()` takes no arguments, and if the value of the object is in $\{0, \ldots, k-1\}$, then the operation increments the value and returns the previous value. Otherwise, the value of the object is unchanged and the integer k is returned. Operation `CAS(`old, new`)` takes as argument the integers *old* and *new*. If the value of the object is equal to *old*, then the operation changes the value of the object to *new* and returns `true`. Otherwise, the object is unchanged and `false` is returned. Operation `Read()` simply returns the value of the object.

```
Class Atomic CAScounter_k

    shared:
        $x$: int init 0

Operation inc( )
1 if $x = k$ then return $x$
2 $x \leftarrow x + 1$
3 return $x - 1$

Operation CAS($old, new$)
4 if $x = old \vee new \notin \{0, 1, \ldots, k\}$ then return false
5 $x \leftarrow new$
6 return true

Operation Read( )
7 return $x$
```

Figure 4.1: Sequential Specification of Type $\mathsf{CAScounter}_k$

4.1.2 Randomized Linearizable Implementation of Type $\mathsf{CAScounter}_k$

We now implement object $\mathsf{RCAScounter}_k$ which is a randomized linearizable implementation of type $\mathsf{CAScounter}_k$. The implementation of object $\mathsf{RCAScounter}_k$ is given in Figure 4.2.

A shared `CAS` object `Count` is used to store the value of the counter object, and is initialized to 0. The object provides methods `inc()`, `CAS()` and `Read()`. Since the object is a randomized implementation, we extend the specification of the object. Specifically, the `inc()` method is allowed to *fail*, in which case the operation does not change the object state, and returns $\perp$ to indicate the failure.

During the `inc()` method, a process p first makes a guess at the counter's current value by rolling a $(k+1)$-sided dice (in line 8) that returns a value in $\{0, \ldots, k\}$ uniformly at random, and stores the value in local variable β. If $\beta = k$, then p performs a `Read()`

on Count(in line 10) to verify the correctness of its guess. If p's guess is correct, then it returns k, otherwise it returns $\perp$ (in line 14) to indicate a failed `inc()` method call. If $\beta \in \{0, \ldots, k-1\}$, then p performs a Count.CAS($\beta, \beta+1$) operation (in line 10) in order to verify the correctness of its guess and to increment Count in one atomic step. If p's guess is correct, then the CAS operation succeeds and the `inc()` method returns the previous value. Otherwise the `inc()` method returns $\perp$ (in line 14) to indicate a failed `inc()` method call.

Method `Read()` simply reads the current value of Count using a Count.Read() operation (line 17) and returns the result of the operation. Method `CAS()` takes two integer parameters old, new, and in line 15 performs a safety check, where it checks whether the value of new is in $\{0, \ldots, k\}$. If the safety check fails, then the method simply returns **false**. Otherwise, it attempts to change the value of Count from old to new using the Count.CAS(old, new) operation (in line 16) and returns the result of the operation.

4.1.3 Analysis and Properties of Object RCAScounter$_k$

Consider an instance of the RCAScounter$_k$ object. Let H be an arbitrary history that consists of all method calls on the instance, except failed `inc()` calls and pending calls that are yet to execute line 10 (`Read` operation), line 12 (`CAS` operation), line 16 (`CAS` operation) or line 17 (`Read` operation). If a failed `inc()` is in the history, it can be linearized at an arbitrary point between its invocation and response, as it does not affect the validity of any other operations. Therefore, it suffices to prove that the history without failed `inc()` operations is linearizable, and then linearizability of the original history follows. The same argument applies to omitting the selected pending method calls. Since the selected pending method calls do not change any shared object, they cannot affect the validity of any other operations.

We define a point $pt(u)$ for every method u in H. Let $I(u)$ be the interval between u's

```
Class RCAScounter_k

    shared:
        Count: int init 0
    local:
        $\beta$: int init 0

Method inc( )
    $\beta \leftarrow$ random$(0, 1, \ldots, k)$
    if $(\beta = k)$ then
        if (Count.Read() $= k$) then return $k$
    else
        if Count.CAS$(\beta, \beta + 1)$ then return $\beta$
    end
    return $\perp$

Method CAS($old, new$)
    if $new \notin \{0, 1, \ldots, k\}$ then return false
    return Count.CAS($old, new$)

Method Read( )
    return Count.Read()
```

Figure 4.2: Implementation of Object RCAScounter$_k$

invocation and response. Let S be the sequential history obtained by ordering the method calls in H according to the points $pt(u)$. To show that $\mathsf{RCAScounter}_k$ is a randomized linearizable implementation of the type $\mathsf{CAScounter}_k$, we need to show that the sequential history S is valid, i.e., S lies in the specification of type $\mathsf{CAScounter}_k$ object, and that $pt(u)$ lies in $I(u)$. Let $\mathcal{C}$ be an object of type $\mathsf{CAScounter}_k$, and let S_v be the sequential history obtained when the operations of S are executed sequentially on object $\mathcal{C}$ in the order as given in S. Clearly, S_v is a valid sequential history in the specification of type $\mathsf{CAScounter}_k$ by construction. Then to show that S is valid, we show that $S = S_v$.

Lemma 4.1.1. *Object* $\mathsf{RCAScounter}_k$ *is a randomized linearizable implementation of type* $\mathsf{CAScounter}_k$*.*

Proof. Let A be an instance of the $\mathsf{RCAScounter}_k$ object. Consider an arbitrary history H that consists of all completed method calls on A, except failed `inc()` calls, and all pending method calls on A that have executed a successful `CAS` operation. We now define point $pt(u)$ for every method u in H.

If u is a `Read()` method call then define $pt(u)$ to be the point in time when the `Read` operation in line 17 is executed.

If u is an `inc()` method call that returns from line 10 then $pt(u)$ is the point in time of the `Read` operation in line 10, and if u's `CAS` operation in line 12 succeeds then $pt(u)$ is the point in time of the `CAS` operation in line 12. By construction, a `Read` or `CAS` operation has been executed during every `inc()` call in H, and no failed `inc()` calls are in H. Then it follows that we have defined $pt(u)$ for every `inc()` call u in H.

If u is a `CAS()` method call that returns from line 15 then $pt(u)$ is any arbitrary point during $I(u)$, and if u returns from line 16 then $pt(u)$ is the point in time of the `CAS` operation in line 16.

Clearly $pt(u) \in I(u)$ for every method u in H.

Let u_i be the i-th operation in S and v_i be the i-th operation in S_v. Let $\mathsf{Count}(u_i)^+$ denote the value of object Count immediately after $pt(u_i)$, and let $\mathcal{C}(v_i)^+$ denote the value of object $\mathcal{C}$ after operation v_i in S_v. We assume that u_0 is a method call that does not change the state of any shared object of instance A (such as a `Read()` method) and returns the initial value of the object. This assumption can be made without loss of generality, because the removal of a method call that does not change the state of the object from a linearizable history always leaves a history that is also linearizable. The purpose of the assumption is to simplify the base case of our induction hypothesis.

We now prove by induction on integer i, that $\mathsf{Count}(u_i)^+ = \mathcal{C}(v_i)^+$, and that the return value of u_i matches the value returned by v_i, thereby proving $S = S_v$.

Basis $(i = 0)$ Since initially the value of object Count and the value of the atomic $\mathsf{CAScounter}_k$ object is 0, it follows from the definition of the method call u_0, that $\mathsf{Count}(u_0)^+ = \mathcal{C}(v_0)^+ = 0$, and the return value of u_0 matches that of v_0.

Induction Step $(i > 0)$ From the induction hypothesis, $\mathsf{Count}(u_{i-1})^+ = \mathcal{C}(v_{i-1})^+$.

Case a - u_i is an `inc()` method call that executes a successful `CAS()` operation in line 12. Then $pt(u_i)$ is when object Count is incremented from β to $\beta + 1$ by a successful Count.`CAS(`$\beta, \beta + 1$`)` operation in line 12, and thus $\mathsf{Count}(u_{i-1})^+ = \beta$ holds. Also, u_i returns $\beta = \mathsf{Count}(u_{i-1})^+$. Since u_i fails the if-condition of line 9, $\beta = k$ and therefore $\mathsf{Count}(u_{i-1})^+ = \beta = k$ holds. Now consider operation v_i in S_v. Since $\mathcal{C}(v_{i-1})^+ = \mathsf{Count}(u_{i-1})^+ = k$, the if-condition of line 1 fails, and the value of the atomic $\mathsf{CAScounter}_k$ is incremented in line 2 and $\mathcal{C}(v_{i-1})^+$ returned in line 3. Hence $\mathsf{Count}(u_i)^+ = \mathcal{C}(v_i)^+$ and the return values match.

Case b - u_i is an `inc()` method call that returns from line 10. Then $pt(u_i)$ is when the `Read()` operation on the object Count is executed in line 10. Clearly, the value returned by the `Read()` operation on the object Count at $pt(u_i)$ is $\mathsf{Count}(u_{i-1})^+$. Since the if-condition of line 10 is satisfied, $\mathsf{Count}(u_{i-1})^+ = k$ and u_i returns integer k without changing object Count. Now consider operation v_i in S_v. Since $\mathcal{C}(v_{i-1})^+ = \mathsf{Count}(u_{i-1})^+$ and $\mathsf{Count}(u_{i-1})^+ = k$, the if-condition of line 1 is satisfied and integer k is returned without changing the atomic $\mathsf{CAScounter}_k$ object. Hence $\mathsf{Count}(u_i)^+ = \mathcal{C}(v_i)^+$ and the return values match.

Case c - u_i is a `CAS()` method call that returns from line 15. Then the if-condition of line 15 is satisfied and thus $new \notin \{0, 1, \ldots, k\}$ and u_i returns **false** without changing Count. Now consider operation v_i in S_v. Since $new \notin \{0, 1, \ldots, k\}$, the if-condition of line 4 will be satisfied and the Boolean value **false** is returned without changing the value of object $\mathcal{C}$. Hence $\mathsf{Count}(u_i)^+ = \mathcal{C}(v_i)^+$ and the return values match.

Case d - u_i is a `CAS()` method call that returns from line 16. Then $pt(u_i)$ is when the `CAS` operation on the object Count is executed in line 16, and u_i returns the result of this `CAS` operation. The `CAS` operation attempts to change the value of Count from *old* to *new*, therefore if $\mathsf{Count}(u_{i-1})^+ = old$ then $\mathsf{Count}(u_i)^+ = new$ and u_i returns **true**, or else Count remains unchanged and u_i returns **false**. Now consider operation v_i in S_v. From the code structure, if $\mathcal{C}(v_{i-1})^+ = old$ then $\mathcal{C}(v_i)^+ = new$ and the Boolean value **true** is returned. And if $\mathcal{C}(v_{i-1})^+ = old$ then the value of object $\mathcal{C}$ remains unchanged and the Boolean value **false** is returned. Hence $\mathsf{Count}(u_i)^+ = \mathcal{C}(v_i)^+$ and the return values match. $\square$

Lemma 4.1.2. *The probability that an* `inc()` *method call returns* $\perp$ *is* $k/(k+1)$ *against the weak adversary.*

Proof. Let the process calling the `inc()` method call (say u) be p and let the value of the object Count immediately before p executes line 8 be z. Since the adversary is weak, no other process executes a shared memory operation after p chooses β in line 8 and before p finishes executing its next shared memory operation. From the code structure,

p returns $\perp$ during u (in line 14) if and only if $z = \beta$. Since

$$\text{Prob}(z = \beta) = 1 - \text{Prob}(z = \beta) = 1 - \frac{1}{k+1} = \frac{k}{k+1},$$

the claim follows. $\square$

The following claim follows immediately from an inspection of the code.

Lemma 4.1.3. *Each of the methods of* $\mathsf{RCAScounter}_k$ *has step complexity* $\mathcal{O}(1)$*, and is wait-free.*

The following theorem follows from Lemmas 4.1.1-4.1.3.

Theorem 4.1.1. *Object* $\mathsf{RCAScounter}_k$ *is a randomized wait-free linearizable implementation of type* $\mathsf{CAScounter}_k$*, where the probability that an* `inc()` *method call fails is* $\frac{k}{k+1}$ *against the weak adversary. Each of the methods of* $\mathsf{RCAScounter}_k$ *has step complexity* $\mathcal{O}(1)$.

4.2 Single-Fast-Multi-Slow Universal Construction

We wish to implement a wait-free *universal construction* object for n processes that ensures some operations are performed in $\mathcal{O}(1)$ steps and no operation takes more than $\mathcal{O}(n)$ steps. A universal construction object provides a linearizable concurrent implementation of any object with a sequential specification that can be given by deterministic code. In our abortable lock, we use an arbitration tree, and at every node of that arbitration tree we wish to use a concurrent implementation of an object that supports some operations executable in $\mathcal{O}(1)$ steps and some other operations executable in $\mathcal{O}(n)$ steps, where n is the maximum number of processes that can access the object concurrently. We now establish the properties of our universal construction object $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$ that implements an object O of type T.

Object $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$ provides two methods, `performFast(`op`)` and `performSlow(`op`)`, to perform any operation on object O, where op is the complete description of the operation to be performed. Method call `performFast(`op`)` is called a *fast* operation, while method call `performSlow(`op`)` is called a *slow* operation. Object $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$ is restricted in its usage in that no two processes are allowed to execute fast operations concurrently. Method `performFast()` has $\mathcal{O}(1)$ step complexity and method `performSlow()` has $\mathcal{O}(n)$ step complexity.

In this section, rather than implementing object $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$, we implement a *lock-free* universal construction object $\mathsf{SFMSUnivConstWeak}\langle\mathsf{T}\rangle$, with slightly weaker properties than $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$. An object implementation is lock-free, if in any infinite history H where processes continue to take steps, and H contains only operations on that object, some operation finishes. Object $\mathsf{SFMSUnivConstWeak}\langle\mathsf{T}\rangle$ has the same properties as object $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$ except method `performFast()` is lock-free with unbounded step-complexity.

There is a standard technique called *operation combining* [40] that can be applied to transform our lock-free object SFMSUnivConstWeak$\langle$T$\rangle$ to the wait-free object SFMSUnivConst$\langle$T$\rangle$ with $\mathcal{O}(n)$ step complexity for method `performFast()`.

Operation Combining Technique. In principle the technique works as follows: Processes maintain an n-element array, say announce, where process i "owns" slot i, and processes store in their respective slots the operation that they want to apply. When a process p wants to apply an operation it first "announces" its operation by writing the operation to the p-th element of the array. Then p attempts to *help* the "next" operation in the announce array by attempting to apply that operation if it has not been applied, yet. An index to the "next" operation to be applied is maintained in the same register that stores the state of the concurrent object. Every time an announced operation is applied, the index is also incremented modulo n in one atomic step. The response of applied operations is stored in another n-element array, say response, which can sometimes be combined with the announce array. Sequence numbers are used to ensure that an announced operation is not applied more than once. Since the index of the "next" operation cycles the announce array, a process needs to help announced operations $\mathcal{O}(n)$ times before its own announced operation is applied, at which point it can stop.

Herlihy [40] introduced this technique as a general methodology to transform lock-free universal constructions to wait-free ones. Herlihy presents another example [41] that employs the technique of operation combining to transform a lock-free universal construction to a wait-free one, where the step complexity of the method that performs the operation is bounded to $\mathcal{O}(n)$.

By applying the technique of operation combining we can transform our lock-free universal construction SFMSUnivConstWeak$\langle$T$\rangle$ into our wait-free object SFMSUnivConst$\langle$T$\rangle$. We however do not provide a proof of its properties. Doing so would be repeating the same "standard" proof ideas from [40], and would result in increasing the size of the thesis without contributing to the main ideas of this thesis. We do provide proofs (in this section) for our lock-free universal construction SFMSUnivConstWeak$\langle$T$\rangle$, and the proofs illustrate the main idea from this section, i.e., how to achieve a linearizable concurrent implementation with support for a `performFast()` method of $\mathcal{O}(1)$ step complexity. We now present the implementation of object SFMSUnivConstWeak$\langle$T$\rangle$ (see Figure 4.3).

Shared Data. A shared register mReg stores a 4-tuple (m_0, m_1, m_2, m_3). We use the notation mReg$[i]$ to refer to the $(i+1)$-th tuple element, m_i, stored in register mReg. Element mReg[0] stores the state of object O. Element mReg[1] stores the result of the most recent fast operation performed. Elements mReg[2] and mReg[3] store counts of the number of fast and slow operations performed respectively. Initially mReg[0] stores the initial state of O, mReg[1] has value $\perp$ and (mReg[2],mReg[3]) is $(0, 0)$.

A shared register fastOp is used to *announce* a fast operation to be performed in a pair (s_0, s_1). Element fastOp[0] stores the complete description of a fast operation to be performed. Element fastOp[1] stores a sequence number indicating the number of fast operations that have been announced in the past. This sequence number is used by processes to determine whether an announced fast operation is pending execution. Initially fastOp is $(\perp, 0)$. The methods `performFast()` and `performSlow()` make use of two private methods `helpFast()` and `f()` (see Figure 4.3).

Description of the `f()` method. Method `f()` is implemented using the specification provided by type T. The method takes two arguments $state_1$ and op, where $state_1$ is a state of object O and op is the complete description of an operation to be applied on object O. The method computes the new state $state_2$ and the result $result$, when operation op is applied on object O with state $state_1$. The method then returns the pair $(state_2, result)$. Since no shared memory operations are executed during the method,

```
Class SFMSUnivConstWeak⟨T⟩

  shared:
      mReg: int init $(s_0, \bot, 0, 0)$
      fastOp: int init $(\bot, 0)$
  local:
      $state, res, fc, sc, s1, s1, r1, r2, seq$: int init $0$

Method performFast(op)
  $(state, res, fc, sc) \leftarrow$ mReg.Read()
  fastOp $\leftarrow (op, fc + 1)$
  if $\neg$helpFast() then helpFast()
  $(state, res, fc, sc) \leftarrow$ mReg.Read()
  return $res$

Method f($state_1, op$)
  $state_2 \leftarrow$ state generated when $op$ is applied to object O with state $state_1$
  $res \leftarrow$ result when $op$ is applied to object O with state $state_1$
  return $(state_2, res)$

Method helpFast()
  $(s1, r1, fc, sc) \leftarrow$ mReg.Read()
  $(op, seq) \leftarrow$ fastOp.Read()
  if $fc \geq seq$ then return true
  $(s2, r2) \leftarrow$ f($s1, op$)
  return mReg.CAS($(s1, r1, fc, sc), (s2, r2, seq, sc)$)

Method performSlow(op)
  repeat
      $(s1, r1, fc, sc) \leftarrow$ mReg.Read()
      $(s2, r2) \leftarrow$ f($s1, op$)
      if $s2 = s1$ then return $r2$
      helpFast()
  until mReg.CAS($(s1, r1, fc, sc), (s2, r1, fc, sc + 1)$)
  return $r2$
```

Figure 4.3: Implementation of Object SFMSUnivConstWeak⟨T⟩.

the method has 0 step complexity.

Description of the `performFast()` method. Let p be a process that executes `performFast(`op`)`. In line 1, process p first copies the 4-tuple read from register mReg to its local variables $state, res, fc$ and sc. Then p announces the operation op by writing the pair $(op, fc+1)$ to register fastOp in line 2. After announcing the operation, process p *helps* perform the announced operation by calling the private method `helpFast()` in line 3. If the call to `helpFast()` returns `false`, then p concludes that the announced operation may not have been performed yet. In this case p makes another call to `helpFast()` in line 3 to be sure that the announced operation is performed (we prove later that at most two calls to `helpFast()` are required to perform an announced operation). Process p then reads and returns the result of the performed operation stored in mReg[1] in line 4 and 5, respectively. Since method `performFast()` is not executed concurrently (by assumption), the result of p's operation stored in register mReg is not overwritten before the end of p's `performFast(op)` call.

Description of the `helpFast()` method. Let q be a process that calls and executes `helpFast()`. In line 9, process q first copies the 4-tuple read from register mReg into its local variables $s1, r1, fc$ and sc. The value read from mReg[0] constitutes the state of object O, to which q will attempt to apply the announced operation if required. The value read from mReg[1] is the result of the last fast operation performed on object O. The value read from mReg[2] and mReg[3] is the count of the number of fast and slow operations performed respectively. Process q then reads fastOp in line 10 to find out the announced operation op and the announced sequence number seq. Process q then determines whether the announced operation has already been performed, by checking whether seq is less than or equal to fc in line 11. If so, q concludes that operation op has been performed and returns `true`, otherwise it attempts to perform op in lines 12 and 13.

In line 12 process q calls the private method `f()` to compute the new state $s2$ and the result $r2$ when operation op is applied to object O with state $s1$. In line 13, process p attempts to perform op by swapping the 4-tuple $(s1, r1, fc, sc)$ with $(s2, r2, fc+1, sc)$ using a `CAS` operation on mReg. If the `CAS` is unsuccessful then no changes are made to mReg. This can happen only if some other process performs an announced fast operation in line 13 or a slow operation in line 19. The result of the `CAS` operation of line 13 is returned in either case.

Description of the `performSlow()` method. Let p be a process that calls and executes `performSlow(`op`)`. During the method, p repeats the while-loop of lines 15-19 until p is able to successfully apply its operation op. In line 15, process p first copies the 4-tuple read from register mReg to its local variables $s1, r1, fc$ and sc. In line 16 process q calls the private method `f()` to compute the new state $s2$ and the result $r2$ when operation op is applied to object O with state $s1$. In the case that operation op does not cause a state change in object O, i.e., $s1 = s2$, then p returns result $r2$ in line 17. Otherwise p attempts to apply operation op in line 19 by swapping the 4-tuple $(s1, r1, fc, sc)$ with $(s2, r2, fc, sc+1)$ using a `CAS` operation on register mReg. Before attempting to apply its own operation in line 19 p makes a call to `helpFast()` in line 18 to help perform an announced fast operation (if any). On completing the while-loop, p would have successfully applied its operation op, and thus p returns the result of the applied operation in line 20.

4.2.1 Analysis and Proofs of Correctness

Let a `helpFast()` method call that returns `true` in line 13 (on executing a successful `CAS` operation) be called a *successful* `helpFast()`.

Claim 4.2.1. *(a) The value of* $\mathsf{fastOp}[1]$ *changes only in line 2.*

(b) The value of $\mathsf{mReg}[3]$ *increases by one with every successful* `CAS` *operation in line 19 and no other operation changes* $\mathsf{mReg}[3]$.

(c) The value of $\mathsf{mReg}[2]$ *increases with every successful* `CAS` *operation in line 13 (during a successful* `helpFast()`*), and no other operation changes* $\mathsf{mReg}[2]$.

Proof. Part (a) follows immediately from an inspection of the code. Register mReg is changed only when a process executes a successful `CAS` operation in lines 13 or 19. Furthermore, in line 13 $\mathsf{mReg}[3]$ is not changed and in line 19 $\mathsf{mReg}[2]$ is not changed. Since, in line 19 $\mathsf{mReg}[3]$ is incremented Part (b) follows immediately. Now, for a process to execute line 13, the if-condition of line 11 must fail, hence $\mathsf{mReg}[2]$ is increased from its previous value and Part (c) follows. □

Consider an arbitrary history H where processes access an $\mathsf{SFMSUnivConstWeak}\langle\mathsf{T}\rangle$ object but no two `performFast()` method calls are executed concurrently. Since the fast operations are executed sequentially the happens before order on all `performFast()` method calls in H is a total order.

Claim 4.2.2. *Let* u_t *be the* t*-th* `performFast()` *method call in history* H *being executed by process* p_t*. For* $t \geq 1$ *let* α_t *be the point in time when* p_t *executes line 2 during* u_t *and* γ_t *be the point when* p_t *is poised to execute line 4. Let* u_t*'s* helpers *be the processes that call* `helpFast()` *such that the value read by the processes in line 10 is the value written to register* fastOp *at* α_t*. Let* β_t *be the the first point in time when a helper's call to* `helpFast()` *succeeds after* α_t*. Let* $\alpha_0, \beta_0, \gamma_0$ *be the start of execution* H*. Then the following claims hold for all* $t \geq 0$*:*

(S₁) β_t *exists and* β_t *is in* (α_t, γ_t)

(S₂) Throughout (α_t, β_t) *:* $\mathsf{fastOp}[1] = \mathsf{mReg}[2] + 1 = t$

(S₃) Throughout (β_t, α_{t+1}) *:* $\mathsf{fastOp}[1] = \mathsf{mReg}[2] = t$

Proof. We prove claims $(S_1), (S_2)$ and (S_3) by induction over t.

Basis: For $t = 0$, (S_1) and (S_2) are trivially true. By assumption the initial value of $\mathsf{fastOp}[1]$ and $\mathsf{mReg}[2]$ is 0. Consider the interval (β_0, α_1). From Claim 4.2.1(a) it follows that fastOp is written for the first time at α_1. The first point when one of the invariants (S_3) is destroyed is if a process (say p) executes a successful `CAS` operation in line 13 during (β_0, α_1). Then p read the value 0 from register $\mathsf{fastOp}[1]$ in line 10, since the initial value of $\mathsf{fastOp}[1]$ is 0 and $\mathsf{fastOp}[1]$ is written to for the first time at α_1. Since $\mathsf{mReg}[2]$ is never decremented (from Claim 4.2.1(c)) and $\mathsf{mReg}[2]$ initially has value 0, p satisfies the if-condition of line 11 and p's `helpFast()` call returns `true` in line 11. Therefore, p does not execute line 13, which is a contradiction.

Induction Step: For $t \geq 1$:

Proof of (S_1): Consider the interval (α_t, γ_t). To show that (S_1) holds for t, we need to show that $\mathsf{mReg}[2]$ is changed during (α_t, γ_t). Consider p_t's first call to `helpFast()` in line 3 during u_t. From induction hypothesis (S_3) for $t-1$, it follows that $\mathsf{fastOp}[1] = \mathsf{mReg}[2] = t-1$ during (β_{t-1}, α_t). Then p_t reads value $t-1$ from $\mathsf{mReg}[2]$ in line 1 and writes value t to $\mathsf{fastOp}[1]$ in line 2. Since $\mathsf{fastOp}[1]$ is changed only at α_{t+1} after α_t, it follows that p_t reads t from register $\mathsf{fastOp}[1]$ in line 10.

Case a - p_t returns from line 11: Then p_t read a value from $\mathsf{mReg}[2]$ in line 9 that is at least t. Since $\mathsf{mReg}[2] = t-1$ holds immediately before α_t some process changed $\mathsf{mReg}[2]$ in line 13 during (α_t, γ_t). Hence, (S_1) for t holds.

Case b - p_t returns `true` from line 13: Then p_t has changed $\mathsf{mReg}[2]$ and hence (S_1) holds for t.

Case c - p_t returns `false` from line 13: Then some process q changed register mReg after p_t read mReg in line 9. Now, register mReg is written to only in line 13 or line 19 (from an inspection of the code).

Subcase c1 - q changed mReg by executing line 13: Then q has changed $\mathsf{mReg}[2]$

and hence (S_1) holds for t.

Subcase c2 - q changed mReg by executing line 19: Then p_t executes a second call to `helpFast()` in line 3. Let m be the value of mReg[3] read by p_t in line 9. If p_t's second `helpFast()` call satisfies case (a) or (b) then we get that (S_1) holds for t.

If p_t's second `helpFast()` call returns `false` from line 13, then some process changed mReg after p_t read mReg in line 9. If some process changed mReg by executing line 13 then we get that (S_1) holds for t. Then some process changed mReg by executing line 19 after p_t read mReg in line 9 and let r be the first process to do so. Therefore, r changes the value of mReg[3] from m to $m+1$ in line 19. Then r executed line 15 after q executed a successful `CAS` operation in line 19. Then r completed a call to `helpFast()` in line 18 after α_t. Since r reads mReg after α_t, r satisfied the if-condition of line 11 and executed line 13. If r successfully executes the `CAS` operation in line 13 then we get that (S_1) holds for t. Then some process s must have changed mReg after r read mReg in line 9. Since the value of mReg[3] is only incremented (by Claim 4.2.1(b)) and r changes the value of mReg[3] from m to $m+1$, it follows that s changed mReg in line 13 and hence (S_1) holds for t.

Proof of (S_2) **and** (S_3)**:** From (S_1) for t it follows that β_t exists and $\alpha_t < \beta_t < \gamma_t < \alpha_{t+1}$. From the induction hypothesis invariants (S_2) and (S_3) are true until α_t. Now, one of the invariants (S_2) or (S_3) can be destroyed only if some process executes a successful `CAS` operation in line 13 and changes mReg[2]. By definition of β_t, mReg[2] is unchanged during (α_t, β_t). Then invariants (S_2) and (S_3) continue to hold until β_t. Therefore, claim (S_2) holds for t. It still remains to be shown that claim (S_3) holds for t.

Let p be the process that executes a successful `CAS` operation in line 13 and changes mReg[2] at β_t. Since mReg[2] $= t-1$ immediately before β_t and p executes a successful `CAS` operation in line 13 at β_t, $p.fc = t-1$. Then p executed lines 9 and 10 during (α_t, β_t) and $p.seq = t$. Therefore, invariant (S_3) is true immediately after β_t.

Now, assume another process (say q) destroys one of the invariants (S_3) or S_4 by executing a successful `CAS` operation in line 13 during (β_t, α_{t+1}). Then q must have read register mReg[2] and fastOp[1] after β_t, and therefore q must read the value t from both of them. Then q must have satisfied the if-condition of line 11 and returned `true`. Hence, q does not execute line 13, which is a contradiction. Therefore, invariant (S_3) is true up to α_{t+1}, and thus claim (S_3) holds for t. □

Let H' be a history that consists of all completed method calls in H and all pending method calls that executed line 2 (`Write` operation on register fastOp), or which executed a successful `CAS` operation in line 13 or line 19. We omit all other pending method calls, since during those method calls no operations are executed that changes the state of any shared object, and hence those pending method calls cannot affect the validity of any other operation. Therefore, to prove that history H is linearizable it suffices to prove that history H' is linearizable.

For each method call u in H', we define a point $pt(u)$ and an interval $I(u)$. Let $I(u)$ denote the interval between u's invocation and response. If u is a `performSlow()` method call that returns from line 17 then $pt(u)$ is the point in time of the `Read` operation in line 15, otherwise, $pt(u)$ is the point in time of the `CAS` operation in line 19. If u is a `performFast()` method call, let v be a successful `helpFast()` method call such that v's line 13 is executed after u's line 2 and before u returns. We define $pt(u)$ to be the point of the successful `CAS` operation in v's line 13.

Claim 4.2.3. *For every method call u in H, $pt(u)$ exists and lies in $I(u)$.*

Proof. There are two types of method calls in H, `performFast()` and `performSlow()`.

Case a - u is a `performFast()` method call.

From Claim 4.2.2 it follows that exactly one of u's helpers (see Claim 4.2.2 for definition) succeeds and the helper performs a successful `CAS` operation in line 13 at some point in $I(u)$. Therefore, point $pt(u)$ exists and lies in $I(u)$.

Case b - u is a `performSlow()` method call. By definition $pt(u)$ is assigned to a line of u's code, therefore $pt(u)$ exists and lies in $I(u)$. $\square$

Let S be the sequential history obtained by ordering all method calls u in H' according to the points $pt(u)$. To show that $\mathsf{SFMSUnivConstWeak}\langle\mathsf{T}\rangle$ is a linearizable implementation of an object O of type T, we need to show that the sequential history S is valid, i.e., S lies in the specification of type T, and that $pt(u)$ lies in $I(u)$ (already shown in Claim 4.2.3). Let S_v be the sequential history obtained when the operations of S are executed sequentially on object O, as per their order in S. Clearly, S_v is a valid sequential history in the specification of type T by construction. Then to show that S is valid, we show that $S = S_v$.

Let v_t be the t-th operation in S_v and let u_t be the t-th method call in S. Let UC_t^- and UC_t^+ denote the value of $\mathsf{mReg}[0]$ immediately before and after $pt(u_t)$, respectively. Let O_t^- and O_t^+ denote the state of object O immediately before and after operation v_t, respectively. Let α_t and β_t denote the value returned by u_t and v_t, respectively. Define $\mathsf{UC}_0^+ = \mathsf{UC}_1^-$ and $\mathsf{O}_0^+ = \mathsf{O}_1^-$. Define $\alpha_0 = \beta_0 = \perp$.

Claim 4.2.4. *Suppose a process calls method* $\mathtt{f}(x_1, op)$ *and the method returns the value pair* (x_2, y)*. If* $x_1 = \mathsf{O}_t^-$ *then* $x_2 = \mathsf{O}_t^+$ *and* $y = \beta_t$*.*

Proof. By definition, a call to method $\mathtt{f}(x_1, op)$ returns the value pair (x_2, y) such that x_2 is the state of O when operation op is applied to O while at state x_1 and y is the result of the operation. Then if $x_1 = \mathsf{O}_t^-$ then $x_2 = \mathsf{O}_t^+$ and $y = \beta_t$. $\square$

Claim 4.2.5. *For all* $t \geq 1$.

(S_1) $\mathsf{O}_t^+ = \mathsf{O}_{t+1}^-$ *and* $\mathsf{UC}_t^+ = \mathsf{UC}_{t+1}^-$

(S_2) $\mathsf{UC}_t^- = \mathsf{O}_t^-$

(S_3) $\mathsf{UC}^+_{t-1} = \mathsf{O}^+_{t-1}$ *and* $\alpha_{t-1} = \beta_{t-1}$

Proof. **Proof of** (S_1)**:** Since operations in S_v are executed sequentially, it follows that $\mathsf{O}^+_t = \mathsf{O}^-_{t+1}$. We now show that $\mathsf{UC}^+_t = \mathsf{UC}^-_{t+1}$. Assume $\mathsf{UC}^+_t = \mathsf{UC}^-_{t+1}$. Then some process p changed the value of $\mathsf{mReg}[0]$ by executing a successful `CAS` operation in line 13 or line 19 at some point during the interval $(pt(u_t), pt(u_{t+1}))$. By definition, p's successful `CAS` operation in line 13 or line 19 is $pt(u_\ell)$ for some method call u_ℓ where u_ℓ is the ℓ-th method call in H'. Thus, ℓ is an integer and $t < \ell < t+1$ holds, which is a contradiction.

Proof of (S_2) **and** (S_3)**:** We prove (S_2) and (S_3) by induction over t.

Basis $(t = 1)$ **-** By assumption, initially, $\mathsf{mReg}[0]$ is the initial state of O, hence, $\mathsf{UC}^-_1 = \mathsf{O}^-_1$. Hence, (S_2) is true. (S_3) is true trivially.

Induction Step - We assume (S_2) and (S_3) for t are true and prove that (S_2) and (S_3) for $t+1$ are true. From (S_1) we have, $\mathsf{O}^+_t = \mathsf{O}^-_{t+1}$ and $\mathsf{UC}^+_t = \mathsf{UC}^-_{t+1}$. From (S_3) for t we have $\mathsf{UC}^+_t = \mathsf{O}^+_t$. Therefore, it follows that $\mathsf{UC}^-_{t+1} = \mathsf{O}^-_{t+1}$ and thus (S_2) for $t+1$ is true.

To show (S_3) for $t+1$ is true, we need to show $\mathsf{UC}^+_t = \mathsf{O}^+_t$ and $\alpha_t = \beta_t$. By Claim (S_2) for t, $\mathsf{UC}^-_t = \mathsf{O}^-_t$ holds. Let p_t be the process executing u_t.

Case a - u_t is a `performSlow(`op`)` method call: Let x_1 be the most recent value read by p_t from $\mathsf{mReg}[0]$ in line 15 and let (x_2, y) be the value returned when p_t executes line 16. From the code structure, $\alpha_t = y$.

Subcase (a1) - p_t returns from line 17: Then $pt(u_t)$ is the point when p_t executes a successful `Read` operation on register mReg in line 15. Since p satisfies the if-condition of line 17, $x_2 = x_1$. Thus, $\mathsf{UC}^-_t = \mathsf{UC}^+_t = x_1$.

Subcase (a2) - p_t returns from line 20: Then $pt(u_t)$ is the point when p_t executes a successful `CAS` operation on register mReg in line 19. From the definition of a `CAS` operation, it follows that $\mathsf{UC}^-_t = x_1$ and $\mathsf{UC}^+_t = x_2$.

For both subcases **(a1)** and **(a2)**, $x_1 = \mathsf{UC}^-_t = \mathsf{O}^-_t$ holds. Then from Claim 4.2.4 it follows that $x_2 = \mathsf{O}_t^+$ and $y = \beta_t$. Since $x_2 = \mathsf{UC}_t^+$ and $y = \alpha_t$, $\mathsf{O}_t^+ = \mathsf{UC}_t^+$ and $\alpha_t = \beta_t$.

Case b - u_t is a `performFast(`op`)` method call: Then $pt(u_t)$ is the point when a successful `CAS` operation on register mReg is executed in line 13 of method call w where w is the first successful `helpFast()` method call that begins after u_t's line 2 is executed. Let q be the process executing w. Let x_1 be the value read by q from $\mathsf{mReg}[0]$ in line 9 and let (x_2, y) be the value returned when q executes line 12. From the definition of a `CAS` operation, it follows that $\mathsf{UC}_t^- = x_1$ and $\mathsf{UC}_t^+ = x_2$. Since $x_1 = \mathsf{UC}_t^- = \mathsf{O}_t^-$, from Claim 4.2.4 it follows that $x_2 = \mathsf{O}_t^+$ and $y = \beta_{u_t}$. Since $x_2 = \mathsf{UC}_t^+$, it follows that $\mathsf{O}_t^+ = \mathsf{UC}_t^+$.

From Claim 4.2.2 if follows that $\mathsf{mReg}[1]$ is changed exactly once during u_t, specifically at $pt(u_t)$, where q writes the value y to it. Thus, p reads the value y from $\mathsf{mReg}[1]$ in line 4 since p executes line 4 after $pt(u_t)$ (Claim 4.2.2). Therefore, it follows that $\alpha_{u_t} = y = \beta_{u_t}$. □

Lemma 4.2.1. *History H' has a linearization in the specification of* T.

Proof. By Claim 4.2.3, for each method call u in H', $pt(u)$ exists and lies in $I(u)$. Thus, to show that H' is linearizable we only need to show that S lies in the specification of type T. Thus, we need to show that for all $t \geq 1$, the value returned by v_t matches that value returned by u_t. From Claim 4.2.5 (S_3) it follows that for all $t \geq 1$, $\alpha_t = \beta_t$. □

Lemma 4.2.2. *Object* $\mathsf{SFMSUnivConstWeak}\langle\mathsf{T}\rangle$ *is wait-free.*

Proof. Suppose not. I.e., there exists an infinite history H during which processes take steps but no method call finishes. It is clear from an inspection of method `performFast()` and private method `helpFast()`, that both methods are wait-free. Then if H contains steps executed by a process that executes a call to `performFast()` then the `performFast()` method call finishes since processes continue to take steps in history H – a contradiction. Now consider the only other case, where history H contains steps executed by processes only on `performFast()` method calls. Consider a process p that takes steps in history H and fails to complete its `performSlow()` method call. Then during p's execution p reads register mReg in line 15 and fails its `CAS` operation in line 19 during an iteration of the loop of lines 14-19. Now p's `CAS` operation can fail only if some process executes a successful `CAS` operation in line 13 or line 19 between p's `Read()` and `CAS` operation.

Case a - Some process q executes a successful `CAS` operation in line 19. Then q breaks out of the loop of lines 14-19. Since processes continue to take steps in our infinite history H, q eventually returns from its `performSlow()` method call – a contradiction.

Case b - Some process q executes a successful `CAS` operation in line 13. Then q has performed a successful `helpFast()` method call and incremented mReg[2]. Let the value of mReg[2] after the increment be z. Now consider the next iteration of the loop by process p, where p's `CAS` operation in line 19 fails again. Since **Case a** leads to a contradiction, some process r executed a successful `CAS` operation in line 13. Then r read incremented mReg[2] to some value greater than z in line 13. From the code structure of the `helpFast()` method, r failed the if-condition of line 11, and therefore r read seq = fastOp[1] $> z$ in line 10. Since fastOp[1] is incremented only in line 2 during a `performFast()` method call, it follows that a `performFast()` method was called after q incremented mReg[2] to z in line 13. This is a contradiction to the assumption that processes take steps executing only method `performSlow()` during our history H. □

The following theorem follows from Lemma 4.2.1 and 4.2.2.

Theorem 4.2.1. *Object* SFMSUnivConstWeak⟨T⟩ *is a lock-free universal construction object that implements an object* O *of type* T*, for n processes, where n is the maximum number of processes that can access object* SFMSUnivConstWeak⟨T⟩ *concurrently and operations on object* O *are performed using either method* `performFast()` *or* `performSlow()`*, and no two processes execute method* `performFast()` *concurrently. Method* `performFast()` *has* $\mathcal{O}(1)$ *step complexity.*

On applying the standard technique of operation combining [40] to object $\mathsf{SFMSUnivConstWeak}\langle\mathsf{T}\rangle$ we obtain object $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$. The following theorem summarizes the properties of object $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$.

Theorem 4.2.2. *Object* $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$ *is a wait-free universal construction object that implements an object* O *of type* T*, for* n *processes, where* n *is the maximum number of processes that can access object* $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$ *concurrently and operations on object* O *are performed using either method* `performFast()` *or* `performSlow()`*, and no two processes execute method* `performFast()` *concurrently. Method* `performFast()` *has* $\mathcal{O}(1)$ *step complexity and method* `performSlow()` *has* $\mathcal{O}(n)$ *step complexity.*

4.3 $\mathsf{AbortableProArray}_k$: An Abortable Promotion Array Type

We now present the type of an object we call Abortable Promotion Array, which is a key part of the construction of our abortable lock in Section 4.4. The universal construction object $\mathsf{SFMSUnivConst}\langle\rangle$ is used to get a concurrent wait-free linearizable implementation of such an object. An object O of type $\mathsf{AbortableProArray}_k$ stores a vector of k integer pairs. Initially the value of O is $(\langle 0, \perp\rangle, \ldots, \langle 0, \perp\rangle)$. Let the $(i+1)$-th value of O be denoted by $O[i]$. In the context of our abortable lock, the i-th element of the array stores the current state of process with ID i, and a sequence number associated with the state.

The type $\mathsf{AbortableProArray}_k$ supports operations `collect()`, `abort()`, `promote()`, `remove()` and `reset()` (see Figure 4.4). Operation `collect(`X`)` takes as argument an array $X[0 \ldots k{-}1]$ with values in $\{\perp\} \cup \mathbb{Z}$. For all i in $\{0, \ldots, k-1\}$, if $O[i] = \langle \mathsf{ABORT}, s\rangle$ and $X[i] = \perp$ then the operation changes $O[i]$ to value $\langle \mathsf{REG}, X[i]\rangle$, for some $s \in \mathbb{Z}$, where REG and ABORT are constants set to value 1 and 3 respectively. Otherwise the value of $O[i]$ is unchanged. In the context of our abortable lock, operation `collect()` is used to "register" processes in the array. Process i is said to be *registered* in the array if a `collect()` operation changes $O[i]$ to value $\langle \mathsf{REG}, s\rangle$, for some $s \in \mathbb{Z}$. Then at most k processes can be registered with a `collect()` operation. The object also allows individual processes to "abort" themselves from the array using the operation `abort()`. Operation `abort(`i, s`)` takes as argument the integers i and seq, where $i \in \{0, \ldots, k-1\}$ and $s \in \mathbb{Z}$. The operation changes $O[i]$ to value $\langle \mathsf{ABORT}, s\rangle$ and returns `true`, only if $O[i]$ is not equal to $\langle \mathsf{PRO}, s\rangle$, for some $s \in \mathbb{Z}$, where PRO is a constant set to value 2. Otherwise the operation returns `false`. In the context of our abortable lock, operation `abort(`i, s`)` is executed only by process i, in order to "abort" itself from the array. Process i is said to *abort* from the array if it executes an `abort(`i, s`)` operation that returns `true`. A registered process in the array that has not aborted is "promoted" using the `promote()`

```
Class Atomic AbortableProArray$_k$

  shared:
      $A$: array of int pairs init $\langle \bot, \bot \rangle$
      REG, PRO, ABORT: const int $1, 2, 3$ respectively

Operation reset()
1  for $i \leftarrow 0$ to $k-1$ do
2      $A[i] \leftarrow \langle 0, \bot \rangle$
3  end

Operation abort(int $i$, int $seq$)
4  $\langle v, s \rangle \leftarrow A[i]$
5  if $v =$ PRO then return false
6  $A[i] \leftarrow \langle \text{ABORT}, seq \rangle$
7  return true

Operation collect(int[] $X$)
8  for $i \leftarrow 0$ to $k-1$ do
9      $\langle v, s \rangle \leftarrow A[i]$
10     if $v =$ ABORT $\wedge X[i] = \bot$ then
11         $A[i] \leftarrow \langle \text{REG}, X[i] \rangle$
12     end
13 end

Operation promote()
14 for $i \leftarrow 0$ to $k-1$ do
15     $\langle v, s \rangle \leftarrow A[i]$
16     if $v =$ REG then
17         $A[i] \leftarrow \langle \text{PRO}, s \rangle$
18         return $\langle i, s \rangle$
19     end
20 end
21 return $\langle \bot, \bot \rangle$

Operation remove(int $i$)
22 $\langle v, s \rangle \leftarrow A[i]$
23 $A[i] \leftarrow (\text{ABORT}, s)$
```

Figure 4.4: Sequential Specification of Type AbortableProArray$_k$

operation. Operation `promote()` takes no arguments, and changes the value of the first element (from left to right) in O with value $\langle \mathsf{REG}, s\rangle$, for some $s \in \mathbb{Z}$, to value $\langle \mathsf{PRO}, s\rangle$, and returns $\langle i, s\rangle$, where i is the index of that element. If there exists no element in O with value $\langle \mathsf{REG}, s\rangle$, for some $s \in \mathbb{Z}$, then O is unchanged and the value $\langle \bot, \bot\rangle$ is returned. In the context of our abortable lock, operation `promote()` is executed by some process, in order to "promote" some process registered in the array. Process i is said to be *promoted* if a `promote()` operation returns $\langle i, s\rangle$, for some $s \in \mathbb{Z}$.

Note that an aborted process in the array, cannot be registered into the array or promoted. The abort mechanism is used by processes as an exit strategy out of the array. If a process tries to abort itself from the array but finds that it has already been promoted, then the process is not allowed to abort. This ensures that a promoted process takes responsibility for some activity that other processes expect of it. The entire array can be reset to its initial state using a `reset()` operation.

In our abortable lock of Section 4.5, we need a concurrent wait-free linearizable implementation of type $\mathsf{AbortableProArray}_{\Delta}$, where Δ is the maximum number of processes that can access the object concurrently. We achieve this using the object $\mathsf{SFMSUnivConst}\langle\mathsf{AbortableProArray}_{\Delta}\rangle$, and therefore an operation of type $\mathsf{AbortableProArray}_{\Delta}$ executed using the `performFast()` method has step complexity $\mathcal{O}(1)$, while an operation executed using the `performSlow()` method has step complexity $\mathcal{O}(\Delta)$.

4.4 Array based Randomized Abortable Lock

4.4.1 Introduction

In this section we specify, implement, and prove properties of an object RandALockArray, which is an array based randomized implementation of type TransferableAbortableLock. Type TransferableAbortableLock provides methods `lock()` and `release()` that can be accessed by at most $n + 1$ processes concurrently, while N is the maximum number of processes in the system.

In our abortable lock (that we present in Section 4.5), we use an arbitration tree, and at every node of that arbitration tree we desire a lock object that is abortable. If Δ is the branching factor of the tree, then we wish to design an abortable lock where $\Delta + 1$ is the maximum number of processes that can access the lock of a node concurrently. (The upper bound of $\Delta + 1$ as opposed to Δ is due to a subtlety, and we postpone its discussion to Section 4.5.) We now specify type TransferableAbortableLock.

Method `lock()` takes a single argument, pseudo-ID, with value in $\{0, \ldots, n-1\}$. We denote a `lock()` method call with argument i as $\texttt{lock}_i\texttt{()}$, but refer to $\texttt{lock}_i\texttt{()}$ as `lock()` whenever the context of the discussion is not concerned with the value of i. Method `lock()` returns a non-$\perp$ value if a process captures the lock, otherwise it returns a $\perp$ value to indicate a failed `lock()` call. A `lock()` call can fail only if the process executing it aborted during the call. Method `release()` is called to release the lock, and it takes two arguments, a pseudo-ID with value in $\{0, \ldots, n-1\}$ and an integer j. Method $\texttt{release(}j\texttt{)}$ returns **true** if and only if there exists a concurrent call to `lock()` that eventually returns j. Otherwise method $\texttt{release(}j\texttt{)}$ returns **false**. In the context of the abortable lock in 4.5, the information contained in the integer argument j is used to transfer the responsibility of releasing additional lock objects of type AbortableLock, but in the context of this section, the value passed in argument j is irrelevant. We pass process pseudo-IDs as arguments to the methods, since we want to allow the ability for a process to "transfer" the responsibility of releasing the lock to other processes. Specifically, we desire that if a process p executes a successful $\texttt{lock}_i\texttt{()}$ call and becomes the owner of the lock, then p does not have to release the lock itself, if it can find some process q to call $\texttt{release}_i\texttt{()}$ on its behalf. Such a "lock transfer" behavior is not permitted by objects of type AbortableLock (see Specification 2.7.1). An algorithm that accesses an instance of an object of type TransferableAbortableLock must satisfy the following:

Condition 4.4.1. (a) No two $\texttt{lock}_i\texttt{()}$ calls are executed concurrently for the same i, where $i \in \{0, \dots, n-1\}$.

(b) If a process p executes a successful $\texttt{lock}_i\texttt{()}$ call, then some process q eventually executes a $\texttt{release}_i\texttt{()}$ call where the invocation of $\texttt{release}_i\texttt{()}$ happens after the response of $\texttt{lock}_i\texttt{()}$ (assuming the scheduler is such that q continues to make progress until its $\texttt{release}_i\texttt{()}$ call happens).

(c) For every $\texttt{release}_i\texttt{()}$ call, there must exist a unique successful $\texttt{lock}_i\texttt{()}$ call that completed before the invocation of the $\texttt{release}_i\texttt{()}$ call.

Algorithm 38 is an example algorithm that illustrates a safe usage of object RandALockArray for process $p \in \{0, \dots, n-1\}$.

In the rest of the section we implement object RandALockArray of type TransferableAbortableLock, and prove all properties of the object. We first establish the properties that we desire from object RandALockArray. An execution of an algorithm that accesses an instance of an object RandALockArray and ensures that Condition 4.4.1 is satisfied, has the following properties:

(a) Mutual exclusion, starvation freedom, bounded exit, and bounded abort hold.

(b) The abort-way has $\mathcal{O}(n)$ RMR complexity.

Algorithm 38: An algorithm illustrating the usage of RandALockArray for process p

```
   // shared: L: An instance of RandALockArray
   // local: int $j$ init 0
24 while true do
       <Remainder Section>
25     if L.lock_p() = ⊥ then // Entry Section
           <Critical Section>
26         $j \leftarrow$ arbitrary integer
27         L.release_p($j$) // Exit Section
28     end
29 end
```

(c) If a process does not abort during a `lock()` call, then it incurs $\mathcal{O}(1)$ RMRs in expectation during the call, otherwise it incurs $\mathcal{O}(n)$ RMRs in expectation during the call.

(d) If a process' call to `release(`j`)` returns **false**, then it incurs $\mathcal{O}(1)$ RMRs during the call, otherwise it incurs $\mathcal{O}(n)$ RMRs during the call.

4.4.2 High Level Description

Shared Data. Object RandALockArray is shown in Figure 4.5. The object uses an instance of object $\mathsf{RCAScounter}_2$ called Ctr, an instance of type $\mathsf{AbortableProArray}_n$ called PawnSet, an integer array apply of n `CAS` objects and an integer array Role of n read-write registers, and two `CAS` objects Sync1 and Sync2.

Let L be an instance of object RandALockArray. The array apply of n `CAS` objects is used by processes to "register" and "deregister" themselves from lock L, and to notify each other of certain events at lock L. Using `CAS` operations, processes can perform these actions in one atomic step, and the rest of the processes can then determine the action performed by reading the `CAS` object.

The `CAS` objects Sync1 and Sync2 are used by the "releasers" of lock L to synchronize

```
Class RandALockArray

  shared:
      Ctr: RCAScounter$_2$ init $0$
      PawnSet: Object of type AbortableProArray$_n$ init $\varnothing$
      apply: array $[0 \ldots n-1]$ of int init $\perp$
      Role: array $[0 \ldots n-1]$ of int init $\perp$
      Sync1, Sync2: int init $\perp$,
      KING, QUEEN, PAWN, PAWN_P: const int $0, 1, 2, 3$ respectively
      REG, PRO: const int $4, 5$ respectively
  local:
      $s, val, seq, dummy$: int init $\perp$,
      $flag, r$: boolean init false,
      $A$: array $[0 \ldots n-1]$ of int init $\perp$

Method lock$_i$( )

     // If $i$ satisfies the loop condition in line 2, 7, or 14, and $i$
        has received a signal to abort, then $i$ calls abort$_i$()
   1 $s \leftarrow$ getSequenceNo()
   2 await (apply$[i]$.CAS($\langle \perp, \perp \rangle, \langle$REG$, s \rangle$))
   3 $flag \leftarrow$ true
   4 repeat
   5     Role$[i] \leftarrow$ Ctr.inc()
   6     if (Role$[i]$ = PAWN) then
   7         await (apply$[i] = \langle$PRO$, s\rangle \vee$ Ctr.Read() $= 2$)
   8         if (apply$[i] = \langle$PRO$, s\rangle$) then
   9             Role$[i] \leftarrow$ PAWN_P
  10         end
  11     end
  12 until (Role$[i] \in \{$KING, QUEEN, PAWN_P$\}$)
  13 if (Role$[i]$ = QUEEN) then
  14     await (Sync1 $= \perp$)
  15 end
  16 apply[i].CAS($\langle$REG$, s\rangle, \langle$PRO$, s\rangle$)
  17 if Role$[i]$ = QUEEN then return Sync1 else return $\infty$
```

Figure 4.5: Implementation of Object RandALockArray

Method `abort`$_i$( )

```
if $\neg flag$ then  return $\bot$
apply[i].CAS($\langle$REG, $s\rangle$, $\langle$PRO, $s\rangle$)
if Role[$i$] = PAWN then
    if $\neg$PawnSet.abort($i, s$) then
        Role[$i$] $\leftarrow$ PAWN_P
        return $\infty$
    end
else
    if $\neg$Sync1.CAS($\bot, \infty$) then
        return Sync1
    end
    doCollect$_i$()
    helpRelease$_i$()
end
apply[$i$].CAS($\langle$PRO, $s\rangle$, $\langle\bot, \bot\rangle$)
return $\bot$
```

Method `release`$_i$(int j)

```
$r \leftarrow$ false
if Role[$i$] = KING then
    if $\neg$Ctr.CAS($1, 0$) then
        $r \leftarrow$ Sync1.CAS($\bot, j$)
        if $r$ then  doCollect$_i$()
        helpRelease$_i$()
    end
end
if Role[$i$] = QUEEN then
    helpRelease$_i$()
end
if Role[$i$] = PAWN_P then
    doPromote$_i$()
end
$\langle dummy, s\rangle \leftarrow$ apply[$i$]
apply[$i$].CAS($\langle$PRO, $s\rangle$, $\langle\bot, \bot\rangle$)
return $r$
```

Method `doCollect`$_i$()

```
for $k \leftarrow 0$ to $n-1$ do
    $\langle val, seq\rangle \leftarrow$ apply[$k$]
    if $val$ = REG then  $A[k] \leftarrow seq$  else  $A[k] \leftarrow \bot$
end
PawnSet.collect($A$)
```

Method `helpRelease`$_i$()

```
if $\neg$Sync2.CAS($\bot, i$) then
    $j \leftarrow$ Sync1.Read()
    Sync1.CAS($j, \bot$)
    $j \leftarrow$ Sync2.Read()
    Sync2.CAS($j, \bot$)
    PawnSet.remove($j$)
    doPromote$_i$()
end
```

Method `doPromote`$_i$()

```
PawnSet.remove($i$)
$\langle j, seq\rangle \leftarrow$ PawnSet.promote()
if $j = \bot$ then
    PawnSet.reset()
    Ctr.CAS($2, 0$)
else
    apply[$j$].CAS($\langle$REG, $seq\rangle$, $\langle$PRO, $seq\rangle$)
end
```

Figure 4.6: Implementation of Object RandALockArray (continued)

among themselves. The releasers are processes responsible for releasing lock L. The array Role of n read-write registers is used by processes to keep track of their "role" at lock L. We now provide a high level description of the methods calls, describe terminologies, and discuss the use of each of the internal objects as and when we require them.

Registering at lock L. At the beginning of their `lock()` call processes *register* themselves in the apply array by swapping the value REG atomically into their designated slots (apply[i] for process with pseudo-ID i).

Role assumption at lock L. Object RandALockArray is not an arbitration tree based lock, but later on in Section 4.5 we use object RandALockArray as a lock object at every node of an arbitration tree. As is common in arbitration tree locks, we desire that processes compete to become the owner of a node by capturing a lock associated with a node. A simple `CAS` object lock at every node does not suffice due to the following reasons. If a process p captures locks of several nodes on its path up to the root and aborts before capturing the root lock, then it must release all captured locks. Such lock releases potentially incurs RMRs to processes busy-waiting for the node locks without guaranteeing any progress to those processes towards capturing the main lock. The promotion mechanism of [4] doesn't work in the case of an abortable lock, since p does not own the root lock, and thus p cannot promote waiting processes into a sequential promotion queue. To ensure that waiting processes make progress towards capturing the root lock, we desire that p "collects" busy-waiting processes (if any) at a node into an instance of an object of type $\mathsf{AbortableProArray}_n$, PawnSet, using the operation `collect()`. Once the busy-waiting processes are collected into PawnSet, p can identify a busy-waiting process, if present, using the PawnSet.`promote()` operation, while busy-waiting processes themselves can abort using the PawnSet.`abort()` operation. However to identify a busy-waiting process, p may have to read $\mathcal{O}(\Delta)$ registers just to find a busy-waiting process at a node, where Δ is the branching factor of the arbitration tree. This is problematic since our goal is to bound the number of steps during a passage to $\mathcal{O}(\Delta)$ steps, and thus a process cannot collect at more than one node. For this reason we desire that p transfer all unreleased locks that it owns to the first busy-waiting process it can find. And if there are no busy-waiting processes at a node, then p should somehow be able to release the lock in $\mathcal{O}(1)$ steps. This is to ensure that p can continue to release captured locks of nodes where there are no busy-waiting processes, since p may have captured at most $\Delta - 1$ nodes on its path. To achieve this we use an instance of $\mathsf{RCAScounter}_2$, Ctr, to help decide if there are any busy-waiting processes at the node. Initially, Ctr is 0, and processes attempt to increase Ctr using the $\mathsf{Ctr}.\mathtt{inc()}$ operation after having registered in the apply array. Process p attempts to release the lock of a node by first executing a $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation. If the operation fails then some process q must have further increased Ctr from 1 to 2, and thus p can transfer all unreleased locks to q, if q has not aborted itself. If q has aborted, then q can perform the collect at the node for p, since q can afford to incur $\mathcal{O}(\Delta)$ additional steps apart from the $\mathcal{O}(\Delta)$ steps that q itself will incur in order to release all locks of nodes that it owns. If q has not aborted then p can transfer all its unreleased locks to q, thus making sure some process makes progress towards capturing the root lock. Therefore we have distinct *roles* that define the protocol a process follows during an execution of its passage at lock L.

There are four roles that processes can assume at lock L during their passage, namely *king*, *queen*, *pawn* and *promoted pawn*. We motivated the roles of king, queen and pawn in the form of p, q and busy-waiting processes in our discussion above. We describe the protocol associated with each of the roles in more detail shortly. We ensure that at any point in time during the execution, the number of processes that have assumed the role of a king, queen and promoted pawn at lock L, respectively, is at most one, and thus we refer to them as king_L, $\mathsf{queen}_\mathsf{L}$ and $\mathsf{ppawn}_\mathsf{L}$, respectively. As mentioned earlier, a

$\mathsf{RCAScounter}_2$ instance Ctr is used by processes to determine their role at lock L. Recall that $\mathsf{RCAScounter}_2$ is a bounded counter, and returns values in $\{0, 1, 2\}$ (see Section 4.1). Each of these values corresponds to a role at lock L.

During an execution of the algorithm, Ctr can cycle from its initial value 0 to non-0 values and then back to 0, multiple times. We refer to the duration in which Ctr increases from 0 to 1, and is later reset to 0 as a Ctr-cycle. Initially, Ctr is 0, and processes attempt to increase Ctr in one atomic step after having registered in the apply array. The process that increases Ctr from 0 to 1 assumes the role of the king process of lock L, and thus becomes king_L. The process that increases Ctr from 1 to 2 assumes the role of the queen process of lock L, and thus becomes $\mathsf{queen}_\mathsf{L}$. By specification of object $\mathsf{RCAScounter}_2$, the value of Ctr cannot be increased beyond 2, and all processes that attempt to increase Ctr any further, are returned value 2, and thus they assume the role of a pawn process. Pawn processes then busy-wait until they get *promoted* at lock L (a process is said to be promoted at lock L if it is promoted in PawnSet), or until they see the Ctr value decrease, so that they can attempt to increase Ctr again. We ensure that a process repeats an attempt to increase Ctr at most once, before getting promoted.

Busy-waiting in lock L. The king process, king_L, becomes the first owner of lock Lduring the current Ctr-cycle, and can proceed to enter its Critical Section, and thus it does not busy-wait during `lock()`. The queen process, $\mathsf{queen}_\mathsf{L}$, must wait for king_L to finish its Critical Section, and only then can it enter its Critical Section. Then $\mathsf{queen}_\mathsf{L}$ spins on the shared register Sync1, waiting for king_L to write some integer value into Sync1. Process king_L attempts to write an integer j into Sync1 only during its call to `release(`j`)`, after it has executed its Critical Section. As already mentioned before, the pawn processes wait on their individual slots of the apply array for a notification of their promotion.

A *collect* action at lock L. A collect action is defined as the sequence of steps executed by a process during a call to `doCollect()`. During a call to `doCollect()`, the collecting process (say q) iterates over the array apply reading every slot, and then creates a local array A from the values read and stores the contents of A in the PawnSet object in using the operation PawnSet.`collect(A)`.

The key point to note here is that if some other process has aborted itself from the PawnSet object (by writing the special value ABORT = 3 into its slot in PawnSet), then the operation PawnSet.`collect(A)` does not overwrite the aborted process's value in PawnSet. A collect action is conducted at lock L by either king_L during a call to `release()`, or by $\mathsf{queen}_\mathsf{L}$ during a call to `abort()`.

A *promote* action at lock L. A promote action is defined as the sequence of steps executed by a process while executing operation PawnSet.`promote()`, during a call to method `doPromote()`. The operation returns the pseudo-ID of a process that was collected during a collect action, and has not yet aborted from PawnSet(by executing the operation PawnSet.`abort(i)`, where i is the process's pseudo-ID). A promote action is conducted at lock L either by king_L, $\mathsf{queen}_\mathsf{L}$ or $\mathsf{ppawn}_\mathsf{L}$.

Lock *handover* from king_L to $\mathsf{queen}_\mathsf{L}$. As mentioned, the queen process, $\mathsf{queen}_\mathsf{L}$, waits for king_L to finish its Critical Section and then call `release(`j`)`. During king_L's `release(`j`)` call, king_L attempts to swap integer j into register Sync1, that only king_L and $\mathsf{queen}_\mathsf{L}$ access. If $\mathsf{queen}_\mathsf{L}$ has not "aborted", then king_L successfully swaps j into Sync1, and this serves as a notification to $\mathsf{queen}_\mathsf{L}$ that king_L has completed its Critical Section, and that $\mathsf{queen}_\mathsf{L}$ may now proceed to enter its Critical Section.

***Aborting* an attempt at lock L by $\mathsf{queen}_\mathsf{L}$.** Processes are allowed to abort their attempt to capture lock L, if they have received a signal to abort, and they happen to busy-wait in lock L. They do so by abandoning their `lock()` call and executing a call to abort() instead. To abort, $\mathsf{queen_L}$ first changes the value of its slot in the apply array from REG to PRO, to prevent itself from getting collected in future collects. Since $\mathsf{king_L}$ and $\mathsf{queen_L}$ are the first two processes at L, $\mathsf{king_L}$ will eventually try to handover L to $\mathsf{queen_L}$. To prevent $\mathsf{king_L}$ from handing over lock L to $\mathsf{queen_L}$, $\mathsf{queen_L}$ attempts to swap a special value ∞ into Sync1 in one atomic step. If $\mathsf{queen_L}$ fails then this implies that $\mathsf{king_L}$ has already handed over L to $\mathsf{queen_L}$, and thus $\mathsf{queen_L}$ returns from its call to abort() with the value written to Sync1 by $\mathsf{king_L}$, and becomes the owner of L. If $\mathsf{queen_L}$ succeeds then $\mathsf{queen_L}$ is said to have successfully aborted, and thus $\mathsf{king_L}$ will eventually fail to hand over lock L. Since $\mathsf{queen_L}$ has aborted, $\mathsf{queen_L}$ now takes on the responsibility of collecting all registered processes in lock L, and storing them into the PawnSet object. After performing a collect, $\mathsf{queen_L}$ then synchronizes with $\mathsf{king_L}$ again, to perform a promote, where one of the collected processes is promoted. After that, $\mathsf{queen_L}$ deregisters from the apply array by resetting its slot to the initial value $\langle\perp,\perp\rangle$.

***Aborting* an attempt at lock L by a pawn process.** Processes are allowed to abort their attempt to capture lock L, if they have received a signal to abort and are busy-waiting in lock L. They do so by abandoning their lock() call and executing a call to abort() instead. To abort, the pawn process (say p) first changes the value of its slot in the apply array from REG to PRO, to prevent itself from getting collected in future collects. It then attempts to abort itself from the PawnSet object by writing the special value ABORT $= 3$ into its slot in PawnSet atomically using the operation PawnSet.abort(p). If p's attempt is unsuccessful then it implies that p has already been promoted, and thus p can assume the role of a promoted pawn, and thus become the owner of L. In this case, p returns from its call to abort() with value ∞ and becomes the owner of L. If p's attempt is successful then p cannot be collected or promoted in future collects and promotion events. In this case, p deregisters from the apply array by resetting its slot to the initial value $\langle\bot,\bot\rangle$, and returns $\bot$ from its call to `abort()`.

Releasing lock L. Releasing lock L can be thought of as a group effort between the king_L, $\mathsf{queen}_\mathsf{L}$ (if present at all), and the promoted pawns (if present at all). To completely release lock L, the owner of L needs to reset Ctr back to 0 for the next Ctr-cycle to begin. However, the owner also has an obligation to hand over lock L to the next process waiting in line for lock L. We now discuss the individual strategies of releasing lock L, by king_L, $\mathsf{queen}_\mathsf{L}$ and the promoted processes. To release lock L, the owner of L executes a call to `release(`j`)`, for some integer j.

Synchronizing the release of lock L by king_L and $\mathsf{queen}_\mathsf{L}$. Process king_L first attempts to decrease Ctr from 1 to 0 using a `CAS` operation. If it is successful, then king_L was able to end the Ctr-cycle before any process could increase Ctr from 1 to 2. Thus, there was no $\mathsf{queen}_\mathsf{L}$ process or pawn processes waiting for their turn to own lock L, during that Ctr-cycle. Then king_L is said to have released lock L.

If king_L's attempt to decrease Ctr from 1 to 0 fails, then king_L knows that there exists a $\mathsf{queen}_\mathsf{L}$ process that increased Ctr from 1 to 2. Since $\mathsf{queen}_\mathsf{L}$ is allowed to abort, releasing lock Lis not as straight forward as raising a flag to be read by $\mathsf{queen}_\mathsf{L}$. Therefore, king_L attempts to synchronize with $\mathsf{queen}_\mathsf{L}$ by swapping the integer j into the object Sync1 using a Sync1.`CAS(`$\bot, j$`)` operation. Recall that $\mathsf{queen}_\mathsf{L}$ also attempts to swap a special value ∞ into object Sync1 using a Sync1.`CAS(`$\bot, j$`)` operation, in order to abort its attempt. Clearly only one of them can succeed. If king_L succeeds, then king_L is said to have successfully handed over lock L to $\mathsf{queen}_\mathsf{L}$. If king_L fails, then king_L knows that $\mathsf{queen}_\mathsf{L}$ has aborted and thus king_L then tries to hand over its lock to one of the waiting pawn processes. The procedure to hand over lock Lto one of the waiting pawn processes is to execute a collect action followed by a promote action.

The collect action needs to be executed only once during a Ctr-cycle, and thus we let the process (among king_L or $\mathsf{queen}_\mathsf{L}$) that successfully swaps a value into Sync1, execute the collect action.

If king_L successfully handed over L to $\mathsf{queen}_\mathsf{L}$, it collects the waiting pawn processes, so that eventually when $\mathsf{queen}_\mathsf{L}$ is ready to release lock L, $\mathsf{queen}_\mathsf{L}$ can simply execute a promote action. Since there is no guarantee that king_L will finish collecting before $\mathsf{queen}_\mathsf{L}$ desires to execute a promote action, the processes synchronize among themselves again, to execute the first promote action of the current Ctr-cycle. They both attempt to swap their pseudo-IDs into an empty CAS object Sync2, and therefore only one can succeed. The process that is unsuccessful, is the second among them, and therefore by that point the collection of the waiting pawn process must be complete. Then the process that is unsuccessful, resets Sync1 and Sync2 to their initial value $\perp$, and then executes the promote action, where a waiting pawn process is promoted and handed over lock L. If no process were collected during the Ctr-cycle, or all collected pawn processes have successfully aborted before the promote action, then the promote action fails, and thus the owner process resets the PawnSet object, and then resets Ctr from 2 to 0 in one atomic step, thus releasing lock L, and resetting the Ctr-cycle.

The release of lock L by $\mathsf{ppawn}_\mathsf{L}$. If a process was promoted by king_L or $\mathsf{queen}_\mathsf{L}$ as described above, then the promoted process is said to be handed over the ownership of L, and becomes the first promoted pawn of the Ctr-cycle. Since a collect for this Ctr-cycle has already been executed, process $\mathsf{ppawn}_\mathsf{L}$ does not execute any more collects, but simply attempts to hand over lock L to the next collected process by executing a promote action. This sort of promotion and handing over of lock L continues until there are no more collected processed to promote, at which point the last promoted pawn resets the PawnSet object, and then resets Ctr from 2 to 0 in one atomic step, thus releasing lock L, and resetting the Ctr-cycle.

All owner processes also *deregister* themselves from lock L, by resetting their slot in the apply array to the initial value $\langle\perp,\perp\rangle$. This step is the last step of their `release(`j`)` calls, and processes return a boolean to indicate whether they successfully wrote integer j into $\mathsf{Sync1}$ during their `release(`j`)` call. Note that only $\mathsf{king_L}$ could possibly return **true** since it is the only process that attempts to do so, during its `release(`j`)` calls.

4.4.3 Implementation / Low Level Description

We now describe the implementation of our algorithm in detail. (See Figure 4.5 and 4.6). We now describe the method calls in detail and illustrate the use of each of the internal objects as and when we require them.

The `lock()` method. Suppose p executes a call to `lock`$_i$`()`. Process p first receives a sequence number using a call to `getSequenceNo()` in line 1 and stores it in its local variable s. Method `getSequenceNo()` returns integer k on being called for the k-th time from a call to `lock`$_i$`()`. Since calls to `lock`$_i$`()` are executed sequentially, a sequential shared counter suffices to implement method `getSequenceNo()`. Method `getSequenceNo()` is used to return unique sequence number which helps solve the classic ABA problem. The ABA problem is as follows: If a process reads an object twice and reads the value of the object to be 'A' both times, then it is unable to differentiate this scenario from a scenario where the object was changed to value 'B' in between the two reads of the object. Process p then spins on apply[i] in line 2 until p *registers* itself by swapping the value $\langle\mathsf{REG}, s\rangle$ into apply[i] using a `CAS` operation. Processes write the value REG in the apply array to announce their presence at lock L.

Process p then executes the *role-loop*, lines 4-12, until p either increases the value of Ctr to 1 or 2, or until p is notified of its promotion. Process p begins an iteration of the role-loop by calling the Ctr`.inc()` operation in line 5 and stores the returned value into $\mathsf{Role}[i]$. The returned value determines p's current role at lock L. The shared array Role is used by process p to store its role in slot $\mathsf{Role}[i]$, which can later be read to determine the actions to perform at lock L. This is important because we want to allow the behavior of transferring locks. Specifically, to enable a process q to call $\mathtt{release}_i\mathtt{()}$ on behalf of p, q needs to determine p's role at lock L, which is possible by reading $\mathsf{Role}[i]$.

If the $\mathsf{Ctr}.\mathtt{inc()}$ operation in line 5 fails, i.e., it returns $\perp$, then p repeats the role-loop. Such repeats can happen only a constant number of times in expectation (by Claim 4.1.2). If the value returned in line 5 is 0 or 1, then p has incremented the value of Ctr (from the semantics of a $\mathsf{RCAScounter}_2$ object), and it becomes king_L or $\mathsf{queen}_\mathsf{L}$, respectively, and breaks out of the role-loop in line 12.

If p becomes king_L in line 5, then p fails the if-condition of line 13 and proceeds to execute lines 16-17. In line 16, p changes $\mathsf{apply}[i]$ to the value $\langle \mathsf{PRO}, s\rangle$, to prevent itself from getting promoted in future promote actions. In line 17, p returns from its `lock()` call by returning the special value ∞ (a non-$\perp$ value indicating a successful `lock()` call), since p is king_L.

If p becomes $\mathsf{queen}_\mathsf{L}$ in line 5, then p knows that there exists a king process at lock L, and thus $\mathsf{queen}_\mathsf{L}$ proceeds to spin on $\mathsf{Sync1}$ in line 14 awaiting a notification from king_L. Recall that king_L notifies $\mathsf{queen}_\mathsf{L}$ of $\mathsf{queen}_\mathsf{L}$'s turn to own lock L by writing the integer j into $\mathsf{Sync1}$ during a $\mathtt{release(}j\mathtt{)}$ call. Once p receives king_L's notification (by reading a non-$\perp$ value in $\mathsf{Sync1}$ in line 14), p breaks out of the spin loop of line 14, and proceeds to execute lines 16-17. In line 16, p changes $\mathsf{apply}[i]$ to the value $\langle \mathsf{PRO}, s\rangle$, to prevent itself from getting promoted in future promote actions. In line 17, p returns from its `lock()` call by returning the integer value stored in $\mathsf{Sync1}$ (a non-$\perp$ value indicating a successful `lock()` call).

If the value returned in line 5 is 2, then p does not become king_L or $\mathsf{queen}_\mathsf{L}$, and thus p assumes the role of a pawn. Process p then waits for a notification of its own promotion, or, for the Ctr value to decrease from 2, by spinning on $\mathsf{apply}[i]$ and Ctr in line 7. When p

breaks out of this spin lock, it determines in line 8 whether it was promoted by checking whether the value of $\mathsf{apply}[\mathrm{i}]$ was changed to $\langle \mathsf{PRO}, s\rangle$. A process is promoted only by a $\mathsf{king_L}$, $\mathsf{queen_L}$ or a $\mathsf{ppawn_L}$ during their `release()` call. If p finds that it was not promoted, then p is said to have been *missed* during a Ctr-cycle, and thus p repeats the role-loop. We later show that a process gets missed during at most one Ctr-cycle.

If p was promoted, then it writes a constant value $\mathsf{PAWN\_P} = 3$ into $\mathsf{Role}[i]$ in line 9 and becomes $\mathsf{ppawn_L}$. Since p has been promoted, p knows that both $\mathsf{king_L}$ and $\mathsf{queen_L}$ are no longer executing their entry or Critical Section, and thus p owns lock L now. Then p goes on to break out of the role-loop in line 12, and proceeds to return from its `lock()` call by returning the special value ∞ (a non-$\perp$ value indicating a successful `lock()` call), since p is $\mathsf{ppawn_L}$.

The `release()` method. Suppose p executes a call to $\texttt{release}_i(j)$ with an integer argument j. We restrict the execution such that a process calls a $\texttt{release}_i(j)$ method only after a call to a successful $\texttt{lock}_i\texttt{()}$ has been completed.

In line 34, p initializes the local variable r to the boolean value **false**. Local variable r is returned later in line 50 to indicate whether the integer j was successfully written to $\mathsf{Sync1}$ during the release method call. In lines 35, 42 and 45 process p determines its role at the node and the action to perform. In line 49, process p deregisters itself from lock L by swapping $\langle\perp,\perp\rangle$ into $\mathsf{apply}[\mathrm{i}]$. At the end of the method call a boolean is returned in line 50, indicating whether the integer j was written to $\mathsf{Sync1}$.

If p determines that it is $\mathsf{king_L}$, then it attempts to decrease Ctr from 1 to 0 in line 36. This decrement operation will only fail if there exists a queen process at lock L which increased the Ctr to 2 during its `lock()` call. If the decrement operation fails then p has determined that there exists a queen process at lock L and it now synchronizes with $\mathsf{queen_L}$ to perform the collect action. Recall that CAS object $\mathsf{Sync1}$ is used by $\mathsf{king_L}$ and

$\mathsf{queen_L}$ to determine which process performs a collect. In line 37, p attempts to swap integer j into $\mathsf{Sync1}$ by executing a $\mathsf{Sync1}.\mathtt{CAS}(\bot, j)$ operation and stores the result of the operation in local variable r. If p is successful then it performs the collect action by executing a call to $\mathtt{doCollect}_i\mathtt{()}$ in line 38. If p is unsuccessful then it knows that $\mathsf{queen_L}$ will perform a collect. In line 39 p calls the $\mathtt{helpRelease}_i\mathtt{()}$ method to synchronize the release of lock L with $\mathsf{queen_L}$. We describe the method `helpRelease()` shortly.

If p determines that it is $\mathsf{queen_L}$, then it calls the $\mathtt{helpRelease}_i\mathtt{()}$ method call in line 43 to synchronize the release of lock L with $\mathsf{king_L}$.

If p determines that it is a promoted pawn, then it attempts to promote a waiting pawn by making a call to `doPromote()` in line 46.

The `doCollect()` method. Suppose a process p executing a $\mathtt{doCollect}_i\mathtt{()}$ method call. The collect action consists of reading the apply array (left to right), and creating a vector A of n values, where the k-th element is either $\bot$ (to indicate that the process with pseudo-ID k is not a candidate for promotion) or an integer sequence number (to indicate that the process with pseudo-ID k is a candidate for promotion). The vector A is stored in the $\mathsf{AbortableProArray}_n$ instance $\mathsf{PawnSet}$ in line 55 using a $\mathsf{PawnSet}.\mathtt{collect}(A)$ operation. The $\mathsf{PawnSet}.\mathtt{collect}(A)$ operation ensures that if the k-th element of $\mathsf{PawnSet}$ has value $3 = \mathsf{ABORT}$ (written during a $\mathsf{PawnSet}.\mathtt{abort}(k, \cdot)$ operation), then the k-th element is not overwritten during the $\mathsf{PawnSet}.\mathtt{collect}(A)$ operation. This is required to ensure that processes that have expressed a desire to abort are not collected and subsequently promoted.

The `helpRelease()` method. Suppose $\mathsf{king_L}$ calls $\mathtt{helpRelease}_i\mathtt{()}$ and $\mathsf{queen_L}$ calls $\mathtt{helpRelease}_k\mathtt{()}$. During the course of these method calls, $\mathsf{king_L}$ and $\mathsf{queen_L}$ synchronize with each other in order to reset `CAS` objects $\mathsf{Sync1}$ and $\mathsf{Sync2}$, remove themselves from $\mathsf{PawnSet}$, promote a collected process and notify the promoted process. If no process is found in PawnSet that can be promoted, then the PawnSet object is reset to its initial state and Ctr reset to 0. Recall that `CAS` object Sync2 is used as a synchronization primitive by king_L and $\mathsf{queen}_\mathsf{L}$ to determine which process exits last among them, and thus performs all pending release work. In line 56, the process which swaps value i or k into Sync2 by executing a successful `CAS` operation, exits, and the other process performs the pending release work in lines 57 - 63. Let us now refer to this other process as the releasing process. In lines 57 - 58, the releasing process resets Sync1 to its initial value $\perp$. In line 59, the releasing process reads the pseudo-ID written to Sync2 by the exited process (process that executed a successful `CAS` operation on Sync1). The pseudo-ID written to Sync2 is required to remove the exited process from getting promoted in a future promote in case it was collected in PawnSet. In line 61, the releasing process removes the exited process from PawnSet. `CAS` object Sync2 is reset to its initial value $\perp$ in line 60. In line 62, the releasing process calls `doPromote()` to promote a collected process.

The `doPromote()` method. Suppose p executes a call to $\texttt{doPromote}_i\texttt{()}$. In line 64, p removes itself from PawnSet by executing a PawnSet.`remove(`i`)` operation. It does so to prevent itself from getting promoted in case it was collected earlier. In line 65, p performs a promote action by executing a PawnSet.`promote()` operation. If a process was collected and the process has not aborted then its corresponding element (k-th element for a process with pseudo-ID k) in PawnSet will have the value $\langle \mathsf{REG}, \cdot \rangle$. If a process has aborted then its corresponding element in PawnSet will have the value $\langle \mathsf{PRO}, \cdot \rangle$.

If a successful `promote()` operation is executed then an element in PawnSet is changed from $\langle \mathsf{REG}, s \rangle$ to $\langle \mathsf{PRO}, s \rangle$, where $s \in \mathbb{N}$, and the pair $\langle k, s \rangle$ is returned, where k is the index of that element in PawnSet. In this case we say that process with pseudo-ID k was *promoted*. If an unsuccessful `promote()` operation is executed, then no element in PawnSet has the value $\langle \mathsf{REG}, s\rangle$, where $s \in \mathbb{N}$, and thus the special value $\langle \bot, \bot\rangle$ is returned. We then say that no process was promoted. The returned pair is stored in local variables $\langle j, seq\rangle$ in line 65.

If no process was promoted, then p resets PawnSet to its initial value in line 67 using the `reset()` operation, and decreases Ctr from 2 to 0 in line 68. If a process was found and promoted in PawnSet, then that process is notified of its promotion, by swapping its corresponding apply array element's value from REG to PRO using a `CAS` operation in line 70.

Recall that, while executing a `lock()` method call a process may receive a signal to abort. Suppose a process p receives a signal to abort while executing a $\texttt{lock}_i\texttt{()}$ method call. If process p is busy-waiting in lines 2, 7 or 14, then p stops executing $\texttt{lock}_i\texttt{()}$, and instead executes a call to $\texttt{abort}_i\texttt{()}$. If p is poised to execute any line 16 or 17 then it completes its call to $\texttt{lock}_i\texttt{()}$. If p is poised to execute any other line then it continues executing $\texttt{lock}_i\texttt{()}$ until it begins to busy-wait in lines 2, 7 or 14, at which point it stops and calls $\texttt{abort}_i\texttt{()}$. If p does not begin to busy-wait in lines 2, 7 or 14 then it completes its $\texttt{lock}_i\texttt{()}$ call.

The `abort()` method. Suppose p executes a call to $\texttt{abort}_i\texttt{()}$. Process p first determines whether it quit $\texttt{lock}_i\texttt{()}$ while busy-waiting on apply[i] in line 2, and if so, p returns $\bot$ in line 18. If not, then p changes apply[i] to the value PRO in line 19, to prevent itself from getting collected in future collect actions. In line 20, process p determines whether it quit $\texttt{lock}_i\texttt{()}$ while busy-waiting on apply[i] in line 7 or 14, or while busy-waiting on Sync1 in line 14. If p quit while busy-waiting on apply[i] then clearly it is a pawn process, and if it quit while busy-waiting on Sync1 then it is a queen process.

If process p determines that it is a pawn then it attempts to remove itself from PawnSet by executing a PawnSet.$\texttt{abort}(i, s)$ operation in line 21, where s was the sequence number returned in line 1. If p has not been promoted yet, then the operation succeeds and p's corresponding element in $\mathsf{PawnSet}$ is changed to a value $\langle \mathsf{ABORT}, s \rangle$, thus making sure that p can not be collected or promoted anymore. If p has already been promoted then the operation fails and p now knows that it is has been promoted, and assumes the role of a promoted pawn, and in line 22, p writes $\mathsf{PAWN\_P}$ into $\mathsf{Role}[i]$ and returns the special value ∞ in line 23.

If process p determines that it is $\mathsf{queen_L}$ then it first attempts to swap a special value ∞ into $\mathsf{Sync1}$ in line 26 by executing a $\mathsf{Sync1}.\mathtt{CAS}(\perp, \infty)$ operation to indicate its desire to abort. If p is successful then p has determined that it is the first (among $\mathsf{king_L}$ and itself) to exit, and therefore p performs the collect action by calling $\mathtt{doCollect}_i\mathtt{()}$ in line 29. Process p then makes a call to $\mathtt{helpRelease}_i\mathtt{()}$ in line 30 to help release lock L by synchronizing with $\mathsf{king_L}$.

If p was unsuccessful at swapping value ∞ into $\mathsf{Sync1}$ then it knows the $\mathsf{king_L}$ is executing `release()`, and $\mathsf{king_L}$ will eventually perform the collect action. Then p has determined that it is the current owner of lock L, and returns the integer value stored in $\mathsf{Sync1}$ in line 27.

Process p executes line 32 only if p successfully aborted earlier in its `abort()` call, and thus it deregisters itself from lock L by swapping $\langle \perp, \perp \rangle$ into apply[i]. Finally, in line 33, p returns $\perp$ to indicate a successful abort (i.e., a failed `lock()` call).

4.4.4 Analysis and Proofs of Correctness

Let H be an arbitrary history of an algorithm that accesses an instance, L, of object $\mathsf{RandALockArray}$, where Conditions 4.4.1 holds. Then the following claims hold for history H.

Lemma 4.4.1. *Methods* $\mathtt{release}_i(j)$*,* $\mathtt{abort}_i\mathtt{()}$*,* $\mathtt{helpRelease}_i\mathtt{()}$*,* $\mathtt{doCollect}_i\mathtt{()}$*,* $\mathtt{doPromote}_i\mathtt{()}$ *are wait-free.*

Proof. Follows from an inspection of these methods. □

Claim 4.4.1. *No two* $\texttt{release}_i\texttt{()}$ *calls where a shared memory step is pending, are executed concurrently for the same* i*, where* $i \in \{0, \ldots, n-1\}$*.*

Proof. Assume for the purpose of a contradiction that two processes are executing a call to $\texttt{release}_i\texttt{()}$ concurrently for the first time at time t. Then from Condition 4.4.1(b)-(c), it follows that two successful calls to $\texttt{lock}_i\texttt{()}$ were executed before t. From condition 4.4.1(a) it follows that the two successful $\texttt{lock}_i\texttt{()}$ calls did not overlap. Consider the first successful $\texttt{lock}_i\texttt{()}$ call executed by some process p. Since the $\texttt{lock}_i\texttt{()}$ call returned a non-$\perp$ value, the method did not return from line 18. Then p did not abort while busy-waiting in line 2, and thus $\mathsf{apply}[i]$ was set to a non-$\langle\perp,\perp\rangle$ value in line 2 during the first $\texttt{lock}_i\texttt{()}$ call. Let t' be the point in time when $\mathsf{apply}[i]$ was set to a non-$\langle\perp,\perp\rangle$ value in line 2. We now show that the $\mathsf{apply}[i] = \langle\perp,\perp\rangle$ in the duration between $[t', t]$. Suppose not, i.e., some process resets $\mathsf{apply}[i]$ to $\langle\perp,\perp\rangle$ during $[t', t]$. Now, $\mathsf{apply}[i]$ is reset to a $\langle\perp,\perp\rangle$ value only in line 32 during $\texttt{abort}_i\texttt{()}$ or in line 49 during $\texttt{release}_i\texttt{()}$.

Case a - $\mathsf{apply}[i]$ reset to $\langle\perp,\perp\rangle$ in line 49 during $\texttt{release}_i\texttt{()}$. Then the last shared memory step of the $\texttt{release}_i\texttt{()}$ has been executed, and the call has ended for the purposes of the claim. Then the two $\texttt{release}_i\texttt{()}$ calls are not concurrent at t, a contradiction.

Case b - $\mathsf{apply}[i]$ reset to $\langle\perp,\perp\rangle$ in line 32 during $\texttt{abort}_i\texttt{()}$. Since the two $\texttt{lock}_i\texttt{()}$ calls are not concurrent it follows that $\mathsf{apply}[i] = \langle\perp,\perp\rangle$ at the end of the first $\texttt{lock}_i\texttt{()}$ call, and thus $\mathsf{apply}[i]$ is reset to $\langle\perp,\perp\rangle$ in line 32 during the second successful $\texttt{lock}_i\texttt{()}$ call. Now consider the second successful $\texttt{lock}_i\texttt{()}$ call executed by some process q. Then q would repeatedly fail the $\mathsf{apply}[i].\texttt{CAS}(\langle\perp,\perp\rangle,\cdot)$ operation of line 2, and the only way q's $\texttt{lock}_i\texttt{()}$ call could finish, is if q aborts the busy-wait loop of line 2. In which case q executes $\texttt{abort}_i\texttt{()}$, and satisfies the if-condition of line 18 and return $\perp$ in line 18. Then the second $\texttt{lock}_i\texttt{()}$ does not reset $\mathsf{apply}[i]$ in line 32 during $\texttt{abort}_i\texttt{()}$ – a contradiction.

Since $\mathsf{apply}[i] = \langle \perp, \perp \rangle$ throughout $[t', t]$, it then follows from the same argument of **Case b**, that the second $\texttt{lock}_i\texttt{()}$ call is unsuccessful, and thus a contradiction. □

From Claim 4.4.1 and Condition 4.4.1(a) it follows that no two calls to $\texttt{lock}_p\texttt{()}$ or $\texttt{release}_p\texttt{()}$ are executed concurrently for the same p, where $p \in \{0, \dots, n-1\}$. Then we can label the process executing a $\texttt{lock}_p\texttt{()}$ or $\texttt{release}_p\texttt{()}$ call, simply p, without loss of generality. We do so to make the rest of the proofs easier to follow.

Helpful claims based on variable usage.

Claim 4.4.2. *(a)* $\mathsf{Role}[p]$ *is changed by process q, only if $q = p$.*

(b) $\mathsf{Role}[p]$ *is unchanged during* $\texttt{release}_p\texttt{()}$.

(c) $\mathsf{Role}[p]$ *can be set to value* KING, QUEEN *or* PAWN *only when p executes line 5 during* $\texttt{lock}_p\texttt{()}$.

(d) $\mathsf{Role}[p]$ *is set to value* $\mathsf{PAWN\_P}$ *only when p executes line 9 during* $\texttt{lock}_p\texttt{()}$ *or when p executes line 22 during* $\texttt{abort}_p\texttt{()}$.

Proof. All claims follow from an inspection of the code. □

Claim 4.4.3. *(a) The only operations on* $\mathsf{PawnSet}$ *are* $\texttt{collect(}A\texttt{)}$, $\texttt{promote()}$, $\texttt{remove(}i\texttt{)}$, $\texttt{remove(}j\texttt{)}$, $\texttt{abort(}k, s\texttt{)}$ *and* $\texttt{reset()}$ *(in lines 55, 65, 64, 61, 21 and 67, respectively) where A is a vector with values in $\{\perp\} \cup \mathbb{N}$, and $i, j, k \in \{0, 1, \dots, n-1\}$, and $s \in \mathbb{N}$.*

(b) The i-th entry of $\mathsf{PawnSet}$ *can be changed to $\langle \mathsf{REG}, s \rangle = \langle 1, s \rangle$, where $s \in \mathbb{N}$, only when a process executes a* $\mathsf{PawnSet}.\texttt{collect(}A\texttt{)}$ *operation in line 55 where $A[i] = s$.*

(c) The i-th entry of $\mathsf{PawnSet}$ *can be changed to $\langle \mathsf{PRO}, s \rangle = \langle 2, s \rangle$, where $s \in \mathbb{N}$, only when a process executes a* $\mathsf{PawnSet}.\texttt{promote()}$ *operation in line 65.*

(d) *The i-th entry of* PawnSet *can be changed to* $\langle \mathsf{ABORT}, s\rangle = \langle 3, s\rangle$*, where* $s \in \mathbb{N}$*, only when a process executes a* PawnSet.`remove(`i`)`*,* PawnSet.`remove(`j`)` *or* PawnSet.`abort(`k, s`)` *operation in lines 64, 61 or 21, respectively.*

Proof. Part (a) follows from an inspection of the code. Parts (b), (c) and (d) follow from Part (a) and the semantics of type $\mathsf{AbortableProArray}_n$. □

Claim 4.4.4. *Let* $s \in \mathbb{N}$*.*

(a) apply$[p]$ *is changed from* $\langle \perp, \perp\rangle$ *to a non-*$\langle \perp, s\rangle$ *value only when process* p *executes a successful* apply$[p]$.`CAS(`$\langle \perp, \perp\rangle, \langle \mathsf{REG}, s\rangle$`)` *operation in line 2.*

(b) apply$[p]$ *is changed to value* $\langle \mathsf{REG}, s\rangle$ *only when process* p *executes a successful* apply$[p]$.`CAS(`$\langle \perp, \perp\rangle, \langle \mathsf{REG}, s\rangle$`)` *operation in line 2.*

(c) apply$[p]$ *is changed to a* $\langle \perp, \perp\rangle$ *value only when* p *executes a successful* apply$[p]$.`CAS(`$\langle \mathsf{PRO}, s\rangle, \langle \perp, \perp\rangle$`)` *operation either in line 32 or line 49.*

Proof. Parts (a), (b) and (c) follow from an inspection of the code. □

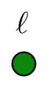
Helpful Notations and Definitions. We now establish a notion of time for our history H. Let the i-th step in H occur at time i. Then every point in time during H is in $\mathbb{N}$.

Let t_p^i denote the point in time immediately after process p has finished executing line i, and no process has taken a step since p has executed the last operation of line i (This operation can be the response of a method call made in line i). Since some private methods are invoked from more than one place in the code, the point in time t_p^i, where i is a line in the method, does not refer to a unique point in time in history H. In those cases we make sure that it is clear from the context of the discussion, which point t_p^i refers to. Let t_p^{i-} denote the point in time when p is poised to execute line i, and no other process takes steps before p executes line i.

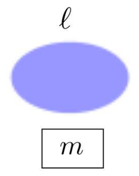
Let p be an arbitrary process and s be an arbitrary integer. We say process p *registers*, when it executes a successful $\mathsf{apply}[p].\mathtt{CAS}(\langle\perp,\perp\rangle,\langle\mathsf{REG},s\rangle)$ operation in line 16. Process p *captures* and *wins* lock L when it returns from $\mathtt{lock}_p\mathtt{()}$ with a non-$\perp$ value. Process p is said to *promote* another process q if p executes a $\mathsf{PawnSet}.\mathtt{promote()}$ operation in line 65 that returns a value $\langle q,s\rangle$, where $s\in\mathbb{N}$. A process p is said to be *promoted* at lock L, if some process q executes a $\mathsf{PawnSet}.\mathtt{promote()}$ operation that returns value $\langle p,s\rangle$, where $s\in\mathbb{N}$.

Process p is said to *hand over* lock L to process q if it executes a successful $\mathtt{CAS}$ operation $\mathsf{L}.\mathsf{Sync1}.\mathtt{CAS}(\perp,j)$ in line 37, where q is the process that last increased Ctr from 1 to 2. Process p is said to have *released* lock L by executing a successful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36, or by executing a successful $\mathsf{Ctr}.\mathtt{CAS}(2,0)$ operation in line 68. Process p either hands over, promotes a process, or releases lock L during a call to $\mathsf{L}.\mathtt{release}_p(j)$ where j is an arbitrary integer. A process *ceases to own* a lock either by releasing lock L or by promoting another process, or by handing over lock L to some other process. Process p is *deregistered* when p executes a successful $\mathsf{apply}[p].\mathtt{CAS}(\langle\mathsf{PRO},s\rangle,\langle\perp,\perp\rangle)$ operation in line 32 or 49. A process p is said to be *not registered in* $\mathsf{PawnSet}$ if the p-th entry of $\mathsf{PawnSet}$ is not value $\langle\mathsf{REG},s\rangle$, where $s\in\mathbb{N}$. The repeat-until loop starting at line 4 and ending at line 12 is called *role-loop*.

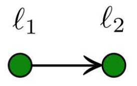
In some of the proofs we use represent an execution using diagrams, and the legend for the symbols used in the diagrams is given in Figure 4.7.



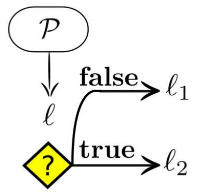
Atomic shared memory operation executed in line ℓ.



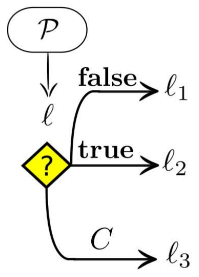
Method call m executed in line ℓ.



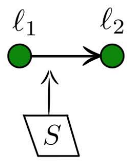
Operation in line ℓ_2 is executed only after the operation in line ℓ_1.



Predicate $\mathcal{P}$ is evaluated in line ℓ.
The next line executed is ℓ_1 if and only if $\mathcal{P}$ is true.
The next line executed is ℓ_2 if and only if $\mathcal{P}$ is false.

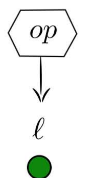


Predicate $\mathcal{P}$ is evaluated in line ℓ.
The next line executed is ℓ_1 if and only if $\mathcal{P}$ is true.
The next line executed is ℓ_2 if and only if $\mathcal{P}$ is false and condition C is false.
The next line executed is ℓ_3 if and only if $\mathcal{P}$ is false and condition C is true.



Statement S holds after line ℓ_1 is executed and before line ℓ_2 is executed.

$S \longrightarrow$ Helpful statement S to improve the readability of the figure.



Operation op executed in line ℓ

Figure 4.7: Legend for Figures 4.8 to 4.16

Releasers of lock and Cease-release events. A process p becomes a *releaser* of lock L at time t when

(R1) p increases Ctr to 1 (i.e., $\mathsf{Ctr}.\mathtt{inc()}$ returns $0 = \mathsf{KING}$) or 2 (i.e., $\mathsf{Ctr}.\mathtt{inc()}$ returns $1 = \mathsf{QUEEN}$), or when

(R2) p is promoted at lock L by some process q .

Claim 4.4.5. *(a) p executes a $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation only in line 36 during $\mathtt{release}_p(j)$.*

(b) p executes a $\mathsf{Sync2}.\mathtt{CAS}(\bot, p)$ operation only in line 56 during p's call to $\mathtt{helpRelease}_p\mathtt{()}$.

(c) p executes a $\mathsf{PawnSet}.\mathtt{promote()}$ operation only in line 65 during p's call to $\mathtt{doPromote}_p\mathtt{()}$.

(d) p executes a $\mathsf{Ctr}.\mathtt{CAS}(2,0)$ operation only in line 68 during p's call to $\mathtt{doPromote}_p\mathtt{()}$.

Proof. All claims follows from an inspection of the code. □

We now define the following *cease-release* events with respect to p :

ϕ_p: p executes a successful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ (at t_p^{36} during $\mathtt{release}_p(j)$).

τ_p: p executes a successful $\mathsf{Sync2}.\mathtt{CAS}(\bot, p)$ (at t_p^{56} during $\mathtt{helpRelease}_p\mathtt{()}$).

π_p: p promotes some process q (at t_p^{65} during $\mathtt{doPromote}_p\mathtt{()}$).

θ_p: p executes an operation $\mathsf{Ctr}.\mathtt{CAS}(2,0)$ (at t_p^{68} during $\mathtt{doPromote}_p\mathtt{()}$).

Process p *ceases* to be a releaser of lock L when one of p's cease-release events occurs. We say process p is a releaser of lock L at any point after it becomes a releaser and before it ceases to be a releaser.

Claim 4.4.6. *(a) Method* $\texttt{doCollect}_p\texttt{()}$ *is called only by process* p *in lines 29 and 38.*

(b) Method $\texttt{helpRelease}_p\texttt{()}$ *is called only by process* p *in lines 39, 43 and 30.*

(c) Method $\texttt{doPromote}_p\texttt{()}$ *is called only by process* p *in line 46 and in line 62 (during* $\texttt{helpRelease}_p\texttt{()}$*).*

(d) If cease-release event ϕ_p *occurs then* p *is executing* $\texttt{release}_p\texttt{(}j\texttt{)}$*.*

(e) If cease-release event τ_p *occurs then* p *is executing* $\texttt{helpRelease}_p\texttt{()}$*.*

(f) If cease-release event π_p *or* θ_p *occurs then* p *is executing* $\texttt{helpRelease}_p\texttt{()}$ *or* $\texttt{doPromote}_p\texttt{()}$*.*

Proof. Parts (a), (b) and (c) follow from an inspection of the code. By definition, cease-release event ϕ_p occurs when p executes a successful $\mathsf{Ctr}.\texttt{CAS(}1,0\texttt{)}$ operation in line 36 during $\texttt{release}_p\texttt{(}j\texttt{)}$, and thus (d) follows immediately. By definition, cease-release event τ_p occurs when p executes a successful $\mathsf{Sync2}.\texttt{CAS(}\perp,p\texttt{)}$ in line 56 during $\texttt{helpRelease}_p\texttt{()}$, and thus (e) follows immediately. By definition, cease-release event π_p occurs only when p executes a $\mathsf{PawnSet}.\texttt{promote()}$ operation that returns a non-$\langle\perp,\perp\rangle$ value in line 65, and cease-release event θ_p occurs only when p executes a $\mathsf{Ctr}.\texttt{CAS(}2,0\texttt{)}$ operation in line 68. Then if cease-release event π_p or θ_p occurs then p is executing $\texttt{doPromote}_p\texttt{()}$. From (c), p could also call $\texttt{doPromote}_p\texttt{()}$ from line 62 during $\texttt{helpRelease}_p\texttt{()}$. Then if cease-release event π_p or θ_p occurs then p is executing $\texttt{doPromote}_p\texttt{()}$ or $\texttt{helpRelease}_p\texttt{()}$. Thus, (f) holds. □

Claim 4.4.7. *Consider* p*'s* k*-th passage, where* $k \in \mathbb{N}$*. Note that* $s = k$*. If* $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ *at some point in time* t *during* p*'s call to* $\texttt{lock}_p\texttt{()}$*, then some process* q *promoted* p *at* t_q^{65} *and* p *became releaser of* L *by condition (R2) at* $t_q^{65} < t$*.*

Proof. From Claim 4.4.2(d), p changes $\mathsf{Role}[p]$ to $\mathsf{PAWN\_P}$ only in line 9 or line 22.

Case a - p changed $\mathsf{Role}[p]$ to $\mathsf{PAWN\_P}$ in line 22: Then p's call to $\mathsf{PawnSet}.\mathtt{abort}(p,s)$ returned `false` in line 21. From the semantics of the $\mathsf{AbortableProArray}_n$ object, it follows that the p-th entry of $\mathsf{PawnSet}$ was set to value $\langle\mathsf{PRO}, s\rangle = \langle 2, s\rangle$. From Claim 4.4.3(c), the p-th entry of $\mathsf{PawnSet}$ is set to value $\langle\mathsf{PRO}, s\rangle$ only when a $\mathsf{PawnSet}.\mathtt{promote()}$ operation returns $\langle p, s\rangle$ in line 65. Then some process q promoted p at t_q^{65} and p became a releaser of L by condition (R2) at $t_q^{65} < t$.

Case b - p changed $\mathsf{Role}[p]$ to $\mathsf{PAWN\_P}$ in line 9: Then p broke out of the spin loop of line 2, and thus $\mathsf{apply}[p] = \langle\mathsf{REG}, s\rangle = \langle\mathsf{PRO}, s\rangle$ at t_p^2. Since p satisfied the if-condition of line 8, it follows that $\mathsf{apply}[p] = \langle\mathsf{PRO}, s\rangle$ at t_p^8. Since p does not change $\mathsf{apply}[p]$ to value $\langle\mathsf{PRO}, s\rangle$ during $[t_p^2, t_p^8]$ it follows that some other process changed $\mathsf{apply}[p]$ to value $\langle\mathsf{PRO}, s\rangle$. Now, $\mathsf{apply}[p]$ is changed to value $\langle\mathsf{PRO}, s\rangle$ by some other process (say q) only in line 70 and thus, from the code structure, q also executed a $\mathsf{PawnSet}.\mathtt{promote()}$ operation that returned $\langle p, s\rangle$ in line 65. Then q promoted p at t_q^{65} and p became a releaser of L by condition (R2) at $t_q^{65} < t$. □

Claim 4.4.8. *Consider p's k-th passage, where $k \in \mathbb{N}$. If $t \in \{\ [t_p^{18-}, t_p^{33}],\ [t_p^{37-}, t_p^{39}],\ [t_p^{43-}, t_p^{43}],\ [t_p^{46-}, t_p^{46}]\ \}$, then cease-release event ϕ_p does not occur before time t.*

Proof. By definition, cease-release event ϕ_p occurs when p executes a successful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36. From Claim 4.4.6(d) cease-release event ϕ_p occurs only during $\mathtt{release}_p(j)$.

Case a - $t \in [t_p^{18-}, t_p^{33}]$: Then p is executing $\mathtt{abort}_p()$ and has not yet executed a call to $\mathtt{release}_p()$. Since cease-release event ϕ_p can occur only during $\mathtt{release}_p()$, cease-release event ϕ_p did not occur before time t.

Case b - $t \in [t_p^{37-}, t_p^{39}]$: Then p must have failed the if-condition of line 36, and thus p executed an unsuccessful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36, and cease-release event ϕ_p did not occur before time t.

Case c - $t \in \{\ [t_p^{43-}, t_p^{43}], [t_p^{46-}, t_p^{46}]\ \}$: From Claim 4.4.9, $\mathsf{Role}[p] \in \{\mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ at t. Since $\mathsf{Role}[p]$ is unchanged during `release`$_p$`()` (Claim 4.4.2(b)), it follows that $\mathsf{Role}[p] = \mathsf{KING}$ at t_p^{35-}. Then p fails the if-condition of line 35, and does not execute line 36 and thus cease-release event ϕ_p did not occur before time t. □

The proof of the following claim has been moved to Appendix A since the proof is long and straight forward.

Claim 4.4.9. *The value of* $\mathsf{Role}[p]$ *at various points in time during p's k-th passage, where* $k \in \mathbb{N}$*, is as follows.*

Time	*Value of* $\mathsf{Role}[p]$
t_p^5	$\{\bot, \mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN}\}$
$[t_p^7, t_p^8]$	PAWN
t_p^9	$\mathsf{PAWN\_P}$
t_p^{13-}	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$
t_p^{14}	QUEEN
$[t_p^{16}, t_p^{17}]$	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$
$[t_p^{19}, t_p^{20-}]$	$\{\mathsf{QUEEN}, \mathsf{PAWN}\}$
t_p^{21}	PAWN
$[t_p^{22}, t_p^{23}]$	$\mathsf{PAWN\_P}$
$[t_p^{26-}, t_p^{30}]$	QUEEN
$[t_p^{34-}, t_p^{35-}]$	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$
$[t_p^{36-}, t_p^{39}]$	KING
t_p^{43-}	QUEEN
t_p^{46-}	$\mathsf{PAWN\_P}$
$[t_p^{49-}, t_p^{50}]$	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$
$[t_p^{51-}, t_p^{55}]$	$\{\mathsf{KING}, \mathsf{QUEEN}\}$
$[t_p^{56-}, t_p^{63}]$	$\{\mathsf{KING}, \mathsf{QUEEN}\}$
$[t_p^{65-}, t_p^{71}]$	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$

Claim 4.4.10. *Consider p's k-th passage, where* $k \in \mathbb{N}$*.*

(a) If process p calls `helpRelease`$_p$`()` *or* `doPromote`$_p$`()` *during* `abort`$_p$`()` *then it does not call* `release`$_p$`(`j`)`*.*

(b) Process p calls `helpRelease`$_p$`()` *at most once.*

(c) Process p calls `doPromote`$_p$`()` *at most once.*

Proof. **Proof of (a):** The following observations follow from an inspection of the code. If p executes `doPromote`$_p$`()` during `abort`$_p$`()`, then it does so during a call to `helpRelease`$_p$`()` in line 62. If p executes `helpRelease`$_p$`()` during `abort`$_p$`()`, then it does so by executing line 30. Then p calls `helpRelease`$_p$`()` or `doPromote`$_p$`()` during `abort`$_p$`()` in line 30 and goes on to return value $\perp$ in line 33. Then p's call to `lock`$_p$`()` returns value $\perp$ and p does not call `release`$_p$`()` (follows from conditions b and c).

Proof of (b): From Part (a), if `helpRelease`$_p$`()` is executed during `abort`$_p$`()` then `release`$_p$`(`j`)` is not executed. Then to prove our claim we need to show that `helpRelease`$_p$`()` is called at most once during `abort`$_p$`()` and `release`$_p$`(`j`)`, respectively. From Claim 4.4.6(b), method `helpRelease`$_p$`()` is called by p only in lines 39, 43 and 30. Since `helpRelease`$_p$`()` is called only once during `abort`$_p$`()` (specifically in line 30), it follows immediately that p executes `helpRelease`$_p$`()` at most once during `abort`$_p$`()`. From Claim 4.4.9, $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ at t_p^{34-}. Since $\mathsf{Role}[p]$ is unchanged during `release`$_p$`()` (Claim 4.4.2(b)), it follows that p satisfies exactly one of the if-conditions of lines 35, 42 and 45, and thus p does not execute both lines 39 and 43. Then p executes `helpRelease`$_p$`()` at most once during `release`$_p$`(`j`)`.

Proof of (c): From Part (a), if `doPromote`$_p$`()` is executed during `abort`$_p$`()` then `release`$_p$`(`j`)` is not executed. Then to prove our claim we need to show that `doPromote`$_p$`()` is called at most once during `abort`$_p$`()` and `release`$_p$`(`j`)`, respectively. From Claim 4.4.6(c), method `doPromote`$_p$`()` is called by p only in line 46 and in line 62 (during `helpRelease`$_p$`()`).

Case a - p called `doPromote`$_p$`()` in line 62 (during `helpRelease`$_p$`()`). Then p is executing `helpRelease`$_p$`()`. From Claim 4.4.6(b), method `helpRelease`$_p$`()` is called by p only in lines 39, 43 and 30. Then p called `helpRelease`$_p$`()` either in line 39, 43 or 30

Case a(i) - p called `helpRelease`$_p$`()` in line 39 or 43 (during `release`$_p$`(`j`)`). Then p is executing `release`$_p$`()`, and since p called `helpRelease`$_p$`()` in lines 39 or 43, p satisfied the if-conditions of lines 35 or 42, and thus $\mathsf{Role}[p] = \mathsf{KING}$ at t_p^{35-} or $\mathsf{Role}[p] = \mathsf{QUEEN}$ at t_p^{42-}, respectively. Since $\mathsf{Role}[p]$ is unchanged during `release`$_p$`()` (Claim 4.4.2(b)), it follows that $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}\}$ during `release`$_p$`()`. Then p fails the if-condition of line 45 and does not execute `doPromote`$_p$`()` in line 46. Hence, p executes `doPromote`$_p$`()` at most once during `release`$_p$`()`.

Case a(ii) - p called `helpRelease`$_p$`()` in line 30. Then p is executing `abort`$_p$`()` and it goes on to return value $\perp$ in line 33. Then p's call to `lock`$_p$`()` returns value $\perp$ and p does not call `release`$_p$`(`j`)` (follows from conditions b and c). Hence, p executes `doPromote`$_p$`()` at most once during `abort`$_p$`()`.

Case b - p called `doPromote`$_p$`()` in line 46. Then p is executing `helpRelease`$_p$`()` and p satisfied the if-condition of lines 45, and thus $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ at t_p^{45-}. Since $\mathsf{Role}[p]$ is unchanged during `release`$_p$`()` (Claim 4.4.2(b)), it follows that $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ during `release`$_p$`()`. Then p failed the if-condition of lines 35 and 42 and p did not execute `helpRelease`$_p$`()` in lines 39 and 43. Hence, p executes `doPromote`$_p$`()` at most once during `release`$_p$`()`. ☐

Claim 4.4.11. *Consider p's k-th passage, where $k \in \mathbb{N}$. Let t be a point in time at which either p is poised to execute* `release`$_p$`(`j`)`*, or $t \in \{\ [t_p^{26-}, t_p^{29}],\ [t_p^{37-}, t_p^{38}],\ [t_p^{51-}, t_p^{55}]$ $,\ t_p^{56-}, [t_p^{57-}, t_p^{62-}],\ t_p^{65-},\ [t_p^{67-}, t_p^{68-}]\ \}$. Then*

(a) none of p's cease-release events have occurred before time t, and

(b) p is a releaser of lock L at time t

Proof. **Proof of (a):** First note that if $t \in [t_p^{51-}, t_p^{55}]$ then p is executing `doCollect()`. From Claim 4.4.6(a), p calls `doCollect()` only in lines 29 and 38. Then if $t \in [t_p^{51-}, t_p^{55}]$ then $t \in [t_p^{29-}, t_p^{29}]$ or $t \in [t_p^{38-}, t_p^{38}]$. Therefore, assume now $t \in [t_p^{26-}, t_p^{29}]$ or $t \in [t_p^{37-}, t_p^{38}]$.

Case a - $t \in \left\{[t_p^{26-}, t_p^{29}], [t_p^{37-}, t_p^{38}]\right\}$: If $t \in [t_p^{26-}, t_p^{29}]$ then from a code inspection, p is executing `abort`$_p$`()` and p did not execute a call to `doPromote`$_p$`()` or `helpRelease`$_p$`()`

before time t. If $t \in [t_p^{37-}, t_p^{38}]$ then p is executing $\texttt{release}_p(j)$ and then from a code inspection and Claim 4.4.10(a) it follows that p did not execute a call to $\texttt{doPromote}_p\texttt{()}$ or $\texttt{helpRelease}_p\texttt{()}$ before time t. Then from Claims 4.4.6(e) and 4.4.6(f) it follows that events τ_p, π_p and θ_p did not occur before time t. Since $t \in [t_p^{26-}, t_p^{29}]$ or $t \in [t_p^{37-}, t_p^{38}]$, it follows from Claim 4.4.8 that cease-release event ϕ_p did not occur before time t.

Case b - $t \in \{\ t_p^{56-},\ [t_p^{57-}, t_p^{62-}]\ \}$: Then p is executing $\texttt{helpRelease}_p\texttt{()}$. Then from Claim 4.4.6(b) it follows that p is executing a call to $\texttt{helpRelease}_p\texttt{()}$ in line 39, 43 or 30. Then from Claim 4.4.8 it follows that cease-release event ϕ_p did not occur before time t. From Claim 4.4.10(b), it follows that this is p's only call to $\texttt{helpRelease}_p\texttt{()}$. From Claim 4.4.6(c), p calls $\texttt{doPromote}_p\texttt{()}$ only in line 46 and in line 62 (during $\texttt{helpRelease}_p\texttt{()}$). Since p has not yet executed line 46 and this is the only call to $\texttt{helpRelease}_p\texttt{()}$, p has not called $\texttt{doPromote}_p\texttt{()}$ before time t. Then from Claim 4.4.6(f) it follows that events π_p and θ_p did not occur before time t. By definition, cease-release event τ_p occurs when p executes a successful $\mathsf{Sync2}.\texttt{CAS(}\bot, p\texttt{)}$ in line 56. If $t = t_p^{56-}$, then clearly cease-release event τ_p did not occur before time t. If $t \in [t_p^{57-}, t_p^{62-}]$, then p satisfied the if-condition of line 56, and thus p executed an unsuccessful $\mathsf{Sync2}.\texttt{CAS(}\bot, p\texttt{)}$ operation in line 56, and thus cease-release event τ_p did not occur before time t.

Case c - $t \in \{\ t_p^{65-},\ [t_p^{67-}, t_p^{68-}]\ \}$: Then p is executing $\texttt{doPromote}_p\texttt{()}$. From Claim 4.4.10(c), it follows that this is the only call to $\texttt{doPromote}_p\texttt{()}$. By definition, cease-release event θ_p occurs only when p executes a $\mathsf{Ctr}.\texttt{CAS(}2, 0\texttt{)}$ operation in line 68 of $\texttt{doPromote}_p\texttt{()}$. Event θ_p did not occur before time t since $t < t_p^{68}$ and this is p's only call to $\texttt{doPromote}_p\texttt{()}$. By definition, cease-release event π_p occurs only when p executes a $\mathsf{PawnSet}.\texttt{promote()}$ operation that returns a non-$\langle\bot, \bot\rangle$ value in line 65 of $\texttt{doPromote}_p\texttt{()}$. If $t = t_p^{65-}$, then cease-release event π_p did not occur before time t since $t_p^{65-} < t_p^{65}$ (and since this is p's only call to $\texttt{doPromote}_p\texttt{()}$). If $t \in [t_p^{67-}, t_p^{68-}]$, then p satisfied the if-condition of line 65, and thus p's $\mathsf{PawnSet}.\texttt{promote()}$ operation returned value $\langle\bot, \bot\rangle$, and thus cease-release event π_p did not occur before time t. Since p calls `doPromote`$_p$`()` only in line 46 and line 62 (during `helpRelease`$_p$`()`), p is executing line 46, 39, 43 or 30. Then from Claim 4.4.8 it follows that cease-release event ϕ_p did not occur before time t.

We now show that cease-release event τ_p did not occur before time t thus completing the proof.

Subcase c(i) - p called `doPromote`$_p$`()` during `helpRelease`$_p$`()`: Then p satisfied the if-condition of line 56, and thus p executed an unsuccessful $\mathsf{Sync2}$.`CAS(`$\perp, p$`)` operation in line 56, and cease-release event τ_p did not occur before time t.

Subcase c(ii) - p called `doPromote`$_p$`()` in line 46: From Claim 4.4.9, $\mathsf{Role}[p] \in \mathsf{PAWN\_P}$ at t_p^{46-}. Since $\mathsf{Role}[p]$ is unchanged during `release`$_p$`()` (Claim 4.4.2(b)), it follows that $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ at t_p^{35-} and t_p^{42-}. Then p fails the if-conditions of lines 35 and 42, and does not execute a call to `helpRelease`$_p$`()` before time t. Then from Claim 4.4.6(e) it follows that cease-release event τ_p did not occur before time t.

Proof of (b): From Part (a), p does not cease to be releaser of L before t. Therefore, to prove our claim we need to show that p becomes a releaser of L at some point $t' < t$. We first show that $\mathsf{Role}[p] \in \{\ \mathsf{KING},\ \mathsf{QUEEN},\ \mathsf{PAWN\_P}\ \}$ at time t. Let t' be the point when p is poised to execute `release`$_p$`(`j`)`. From the inspection of the various points in time chosen for t (including t_p^{34-}, but excluding t') and the table in Claim 4.4.9, it follows that $\mathsf{Role}[p] \in \{\ \mathsf{KING},\ \mathsf{QUEEN},\ \mathsf{PAWN\_P}\ \}$ at time t (including t_p^{34-}, but excluding t'). Clearly $\mathsf{Role}[p]$ is unchanged during $[t', t_p^{34-}]$. Then the value of $\mathsf{Role}[p]$ at t' is the same as that at t_p^{34-}, i.e., $\mathsf{Role}[p] \in \{\ \mathsf{KING},\ \mathsf{QUEEN},\ \mathsf{PAWN\_P}\ \}$.

Case a - $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}\}$ at time t: From Claim 4.4.2(c), $\mathsf{Role}[p]$ is set to KING or QUEEN only when p executes line 5. Then p changed $\mathsf{Role}[p]$ to KING or QUEEN at t_p^5, and thus p became a releaser of lock L by condition (R1) at $t_p^5 = t' < t$.

Case b - $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ at time t: From Claim 4.4.7, it follows that some process q promoted p at t_q^{65} and p became a releaser of L by condition (R2) at $t_q^{65} = t' < t$. $\square$

Claim 4.4.12. *Consider p's k-th passage, where $k \in \mathbb{N}$. If any of process p's cease-release events occurs at time t then p ceases to be the releaser of lock L at time t.*

Proof. To prove our claim we need to show that p is a releaser of L immediately before time t, since by definition p ceases to be a releaser of L when any of p's cease-release events occurs. By definition, cease-release event ϕ_p occurs when p executes a successful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36, cease-release event τ_p occurs when p executes a successful $\mathsf{Sync2}.\mathtt{CAS}(\perp,p)$ in line 56, cease-release event π_p occurs only when p executes a $\mathsf{PawnSet}.\mathtt{promote}()$ operation that returns a non-$\langle\perp,\perp\rangle$ value in line 65, cease-release event θ_p occurs only when p executes a $\mathsf{Ctr}.\mathtt{CAS}(2,0)$ operation in line 68. From Claim 4.4.11(b), p is a releaser of L at t_p^{36-}, t_p^{56-}, t_p^{65-} and t_p^{68-}. Hence, the claim follows. □

We say a process has *write-access* to objects Sync1 and Sync2, respectively, if the process can write a value to Sync1 and Sync2, respectively. We say a process has *registration-access* to object PawnSet, if the process can execute an operation on PawnSet that can write values in $\{\langle a,b\rangle | a \in \{0,1,2\} = \{0,\mathsf{REG},\mathsf{PRO}\}, b \in \mathbb{N}\}$ to some entry of PawnSet. We say a process has *deregistration-access* to object PawnSet, if the process can execute an operation on PawnSet that can write value $\langle\mathsf{ABORT},s\rangle = \langle 3,s\rangle$, where $s \in \mathbb{N}$, to some entry of PawnSet. Object PawnSet is said to be *candidate-empty* if no entry of PawnSet has value $\langle\mathsf{REG},\cdot\rangle$ or $\langle\mathsf{PRO},\cdot\rangle$.

Claim 4.4.13. *Only releasers of L have write-access to* Sync1*,* Sync2 *and registration-access to* PawnSet*.*

Proof. The following observations follow from an inspection of the code. A value can be written to Sync1 only in lines 26, 37 and 58. A value can be written to Sync2 only in lines 56 and 60. From the semantics of the $\mathsf{AbortableProArray}_n$ object, only operations `collect()`, `promote()`, and `reset()` can write values in

$\{\langle a,b\rangle | a \in \{0, \mathsf{REG}, \mathsf{PRO}\} = \{0,1,2\}, b \in \mathbb{N}\}$ to PawnSet. From Claim 4.4.3(a), the operations `collect()`, `promote()`, and `reset()` are executed on PawnSet only in lines 55, 65, and 67, respectively.

Suppose an arbitrary process p writes a value to Sync1 or Sync2, or a value in $\{\langle a,b\rangle | a \in \{0,1,2\}, b \in \mathbb{N}\}$ to an entry of PawnSet. From Claim 4.4.11(b), p is a releaser of L at t_p^{26-}, t_p^{37-}, t_p^{58-}, t_p^{56-}, t_p^{60-}, t_p^{65-}, t_p^{67-} and t_p^{55-}. Hence, the claim follows. □

Claim 4.4.14. *The i-entry of* PawnSet *can be changed only by process i or a releaser of* L*.*

Proof. The values that can be written to PawnSet are in $\{\langle a,b\rangle | a \in \{0,1,2,3\}, b \in \mathbb{N}\}$. A process that can write values in $\{\langle a,b\rangle | a \in \{0,1,2\}, b \in \mathbb{N}\}$ to any entry of PawnSet is said to have registration-access to PawnSet. From Claim 4.4.13 it follows that only a releaser of L has registration-access to PawnSet, therefore only a releaser of L can write values in $\{\langle a,b\rangle | a \in \{0,1,2\}, b \in \mathbb{N}\}$ to the i-th entry of PawnSet. From Claim 4.4.3(d) the value $\langle \mathsf{ABORT}, s\rangle = \langle 3, s\rangle$, where $s \in \mathbb{N}$, can be written to the i-th entry of PawnSet only when a process executes a `remove(`i`)`, `remove(`i`)` or PawnSet.`abort(`i, s`)` operation in line 64, 61 or 21, respectively. From Claim 4.4.11(b), it follows that a process executing lines 64 and 61 is a releaser of L. Since a PawnSet.`abort(`i, s`)` operation in line 21 is executed only by process i, our claim follows. □

Claim 4.4.15. Sync2 *is changed to a non-$\perp$ value only by a releaser of* L *(say r) in line 56 which triggers the cease-release event τ_r.*

Proof. By definition, cease-release event τ_p occurs when p executes a successful Sync2.`CAS(`$\perp, p$`)` in line 56. From a code inspection, Sync2 is changed to a non-$\perp$ value only when some process (say r) executes a successful Sync2.`CAS(`$\perp, r$`)` operation in line 56. From Claim 4.4.13 it follows that Sync2 is changed only by a releaser of L. Then r is a releaser of L when it changes $\mathsf{Sync2}$ to a non-$\perp$ value in line 56 and doing so triggers the cease-release event τ_r. □

Claim 4.4.16. *A* $\mathsf{PawnSet}.\mathtt{promote()}$ *operation is executed only by a releaser of* L *(say* r*), and if the value returned is non-*$\langle\perp,\perp\rangle$ *the cease-release event* π_r *is triggered.*

Proof. By definition, cease-release event π_p occurs only when p executes a $\mathsf{PawnSet}.\mathtt{promote()}$ operation that returns a non-$\langle\perp,\perp\rangle$ value in line 65. From a code inspection, a $\mathsf{PawnSet}.\mathtt{promote()}$ operation is executed only when some process (say r) executes line 56. From Claim 4.4.13 it follows that $\mathsf{PawnSet}$ is changed only by a releaser of L. Then r is a releaser of L when it executes a $\mathsf{PawnSet}.\mathtt{promote()}$ operation, and if the operation returns a non-$\langle\perp,\perp\rangle$ value then the cease-release event π_r is triggered. □

Claim 4.4.17. *During an execution of* $\mathtt{doPromote}_p\mathtt{()}$ *exactly one of the events* π_p *and* θ_p *occurs.*

Proof. By definition, cease-release event π_p occurs when p executes a $\mathsf{PawnSet}.\mathtt{promote()}$ operation in line 65 that returns a non-$\langle\perp,\perp\rangle$ value, and cease-release event θ_p occurs when p executes a $\mathsf{Ctr}.\mathtt{CAS}(2,0)$ operation in line 68 during $\mathtt{doPromote}_p\mathtt{()}$.

Case a - the $\mathsf{PawnSet}.\mathtt{promote()}$ operation in line 65 returns a non-$\langle\perp,\perp\rangle$ value, and thus cease-release event π_p occurs: Then p fails the if-condition of line 66 and line 68 is not executed. Therefore, cease-release event θ_p does not occur.

Case b - the $\mathsf{PawnSet}.\mathtt{promote()}$ operation in line 65 returns $\langle\perp,\perp\rangle$, and thus cease-release event π_p does not occur: Then p satisfies the if-condition of line 66, and executes a $\mathsf{Ctr}.\mathtt{CAS}(2,0)$ operation in line 68. Hence, cease-release event θ_p occurs. □

Claim 4.4.18. *During an execution of* $\mathtt{helpRelease}_p\mathtt{()}$ *exactly one of the events* τ_p, π_p *and* θ_p *occurs.*

Proof. By Claim 4.4.5, events π_p and θ_p can only occur during p's call to $\texttt{doPromote}_p\texttt{()}$, and cease-release event τ_p occurs when p executes a successful $\textsf{Sync2}.\texttt{CAS(}\bot, p\texttt{)}$ operation in line 56.

Case a - p executes a successful $\textsf{Sync2}.\texttt{CAS(}\bot, p\texttt{)}$ operation in line 56, and thus cease-release event τ_p occurs: Then p fails the if-condition of line 56, and returns immediately from its call to $\texttt{helpRelease}_i\texttt{()}$. Therefore, events π_p and θ_p do not occur.

Case b - p executes an unsuccessful $\textsf{Sync2}.\texttt{CAS(}\bot, p\texttt{)}$ operation in line 56, and thus cease-release event τ_p does not occur. Then p satisfies the if-condition of line 56, and calls $\texttt{doPromote}_p\texttt{()}$ in line 62. From Claim 4.4.17, exactly one of the events π_p and θ_p occurs during p's call to $\texttt{doPromote}_p\texttt{()}$. □

Claim 4.4.19. *The value of* $\textsf{Ctr}$ *can change only when a* $\textsf{Ctr}.\texttt{inc()}$*,* $\textsf{Ctr}.\texttt{CAS(}2,0\texttt{)}$ *or* $\textsf{Ctr}.\texttt{CAS(}1,0\texttt{)}$ *operation is executed in lines 5, 68 or 36.*

Proof. From the semantics of the $\textsf{RCAScounter}_2$ object, if $\textsf{Ctr}$ is increased to value i by a $\textsf{Ctr}.\texttt{inc()}$ operation, then its value was $i-1$ immediately before the operation was executed. Then all claims follow from an inspection of the code. □

Claim 4.4.20. *If the value of* $\textsf{Ctr}$ *changes, it either increases by* 1 *or decreases to* 0*. Moreover its values are in* $\{0,1,2\}$.

Proof. From the semantics of the $\textsf{RCAScounter}_2$ object, a $\textsf{Ctr}.\texttt{inc()}$ operation changes the value of $\textsf{Ctr}$ from i to $i+1$ only if $i \in \{0,1\}$. From Claims 4.4.19, the value of $\textsf{Ctr}$ can change only when a $\textsf{Ctr}.\texttt{inc()}$, $\textsf{Ctr}.\texttt{CAS(}2,0\texttt{)}$ or $\textsf{Ctr}.\texttt{CAS(}1,0\texttt{)}$ operation is executed (in lines 5, 68 or 36). Then it follows that the values of $\textsf{Ctr}$ are in$\{0,1,2\}$. It also follows that the value of $\textsf{Ctr}$ either changes from 0 to 1 and back to 0, or it changes from 0 to 1 to 2 and back to 0. □

$\textsf{Ctr}$-Cycle Interval T. Let $T = [t_s, t_e)$ be a time interval where t_s is a point when $\textsf{Ctr}$ is 0 and t_e is the next point in time when $\textsf{Ctr}$ is decreased to 0. For $i \in \{0,1,2\}$

let $I_i = \{t \in T | \mathsf{Ctr} = i \text{ at } t\}$ and let time $I_i^- = \mathtt{min}(I_i)$ and time $I_i^+ = \mathtt{max}(I_i)$. From Claim 4.4.20, it follows immediately that during T the set $I_i, i \in \{0, 1, 2\}$, forms an interval $[I_i^-, I_i^+]$, and $I_2 = \varnothing$ if and only if Ctr is never increased to 2 during T. Moreover, $t_s = I_0^-$ and I_0 *is immediately followed by* I_1 (i.e., $\mathtt{min}(I_1) = \mathtt{max}(I_0) + 1$). If $I_2 = \varnothing$ then I_2 follows immediately after I_1. The Ctr-cycle interval T ends either at time I_1^+ if $I_2 = \varnothing$, or at time I_2^+ if $I_2 = \varnothing$.

Then it also follows that exactly one process changes Ctr from 0 to 1 during T, and it does so at time I_1^-. Let $\mathcal{K}$ be the process that increases Ctr to 1 at time I_1^-. And if $I_2 = \varnothing$ then exactly one process changes Ctr from 1 to 2 during T, and it does so at time I_2^-. If $I_2 = \varnothing$ let $\mathcal{Q}$ be the process that increases Ctr to 2 at time I_2^-. Let $R(t)$ denote the set of processes that are the releasers of lock L at time $t \in T$.

Claim 4.4.21. *If $R(I_0^-) = \varnothing$ and at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \bot$ and $\mathsf{PawnSet}$ is candidate-empty, then the following holds:*

(a) $\forall_{t \in I_0} : R(t) = \varnothing$ and throughout I_0, $\mathsf{Sync1} = \mathsf{Sync2} = \bot$ and $\mathsf{PawnSet}$ is candidate-empty.

(b) $R(I_1^-) = \{\mathcal{K}\}$ and at time I_1^-, $\mathsf{Sync1} = \mathsf{Sync2} = \bot$ and $\mathsf{PawnSet}$ is candidate-empty.

(c) $\mathcal{K}$ executes lines of code of $\mathtt{lock}_{\mathcal{K}}\mathtt{()}$ starting with line 2 as depicted in Figure 4.8. (A legend for the figure is given in Figure 4.7.)

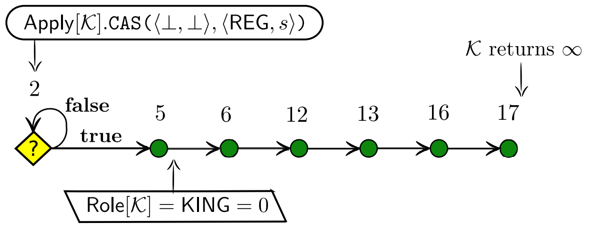

Apply[$\mathcal{K}$].CAS($\langle\bot,\bot\rangle$, $\langle$REG, $s\rangle$)
$\mathcal{K}$ returns ∞
2
false
true
?
5
6
12
13
16
17
Role[$\mathcal{K}$] = KING = 0

Figure 4.8: $\mathcal{K}$'s call to $\mathtt{lock}_{\mathcal{K}}\mathtt{()}$

(d) $\mathcal{K}$'s call to $\mathtt{lock}_{\mathcal{K}}\mathtt{()}$ returns ∞ and $\mathsf{Role}[\mathcal{K}] = \mathsf{KING}$ throughout $[t_{\mathcal{K}}^5, t_{\mathcal{K}}^{17}]$.

(e) $\mathcal{K}$ *executes a* $\mathsf{Ctr}.\texttt{CAS}(1,0)$ *operation in line 36 during* T*, and* $\mathcal{K}$ *does not change* $\mathsf{Sync1}, \mathsf{Sync2}$ *or* $\mathsf{PawnSet}$ *throughout* $[I_1^-, t_{\mathcal{K}}^{36}]$*.*

(f) $\forall_{t \in I_1} : R(t) = \{\mathcal{K}\}$*.*

(g) Throughout I_1*,* $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ *and* $\mathsf{PawnSet}$ *is candidate-empty.*

Proof. **Proof of (a):** Consider the claim $R(t) = \varnothing$ where $t \in I_0$. Since $R(I_0^-) = \varnothing$ holds by assumption, the claim holds at $t = I_0^-$. For the purpose of a contradiction assume the claim fails to hold for the first time at some point t' during I_0. Then some process p becomes a releaser of lock L at time t'. Process p cannot become a releaser of L by p increasing Ctr to 1 or 2 (condition (R1)) at time t', since $\mathsf{Ctr} = 0$ throughout I_0. Therefore, assume it becomes a releaser of L when some process q promotes p (condition (R2)) at t'. By Claim 4.4.12, q ceases to be a releaser of lock L at t'. This is a contradiction to our assumption that p is the first process during I_0 to become a releaser of L.

By assumption the variables $\mathsf{Sync1}, \mathsf{Sync2}$ and $\mathsf{PawnSet}$ are at their initial value at I_0^-. Since the values of these variables are only changed by a releaser of lock L (by Claim 4.4.13) and for all $t \in I_0$, $R(t) = \varnothing$, it follows that the variables are unchanged throughout I_0.

Proof of (b): At time I_1^- Ctr is increased from 0 to 1, and thus the only operation executed is a $\mathsf{Ctr}.\texttt{inc()}$ operation by process $\mathcal{K}$. Then $\mathcal{K}$ becomes a releaser of lock L at time I_1^- by condition (R1). Since for all $t \in I_0$, $R(t) = \varnothing$ (Part (a)), it follows that $R(I_1^-) = \{\mathcal{K}\}$. Since $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty throughout I_0 (Part (a)), and the only operation at time I_1^- is the $\mathsf{Ctr}.\texttt{inc()}$ operation, it follows that $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty at time I_1^- .

Proof of (c) and (d): Since $\mathcal{K}$ is the process that increased Ctr from 0 to 1 at time I_1^-, and since $\mathcal{K}$ can increase Ctr only by executing a $\mathsf{Ctr}.\texttt{inc()}$ operation in line 5 (by Claim 4.4.19) $\mathcal{K}$ set $\mathsf{Role}[\mathcal{K}] = 0 = \mathsf{KING}$ at $t_{\mathcal{K}}^5$. Then from the code structure, $\mathcal{K}$

does not execute lines 7-9, and does not repeat the role-loop, and does not busy-wait in the spin loop of line 14; instead $\mathcal{K}$ proceeds to execute lines 16 - 17 and returns value ∞ in line 17. Since $\mathcal{K}$ does not change $\mathsf{Role}[\mathcal{K}]$ during $[t_{\mathcal{K}}^{5}, t_{\mathcal{K}}^{17}]$, $\mathsf{Role}[\mathcal{K}] = \mathsf{KING}$ throughout $[t_{\mathcal{K}}^{5}, t_{\mathcal{K}}^{17}]$.

Proof of (e): Since $\mathcal{K}$ is the process that increased Ctr from 0 to 1 at time I_1^-, and since $\mathcal{K}$ can increase Ctr only by executing a $\mathsf{Ctr}.\mathtt{inc()}$ operation in line 5 (by Claim 4.4.19) $\mathcal{K}$ set $\mathsf{Role}[\mathcal{K}] = 0 = \mathsf{KING}$ at $t_{\mathcal{K}}^{5}$. From Part (d), $\mathcal{K}$ returns from $\mathtt{lock}_{\mathcal{K}}\mathtt{()}$ with value ∞ in line 17, and thus $\mathcal{K}$ consequently calls $\mathtt{release}_{\mathcal{K}}\mathtt{(}j\mathtt{)}$ (follows from conditions (b) and (c)). Note that $\mathcal{K}$ has not executed any operations on $\mathsf{Sync1}, \mathsf{Sync2}$ and $\mathsf{PawnSet}$ in the process. Then $\mathsf{Role}[\mathcal{K}] = \mathsf{KING}$ at $t_{\mathcal{K}}^{35-}$ and thus p satisfies the if-condition of line 35 and executes the $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36 during T without having executed any operations on $\mathsf{Sync1}, \mathsf{Sync2}$ and $\mathsf{PawnSet}$ in the process. Thus $\mathcal{K}$ did not change $\mathsf{Sync1}, \mathsf{Sync2}$ or $\mathsf{PawnSet}$ during $[I_1^-, t_{\mathcal{K}}^{36}]$.

Proof of (f): Since $R(I_1^-) = \{\mathcal{K}\}$ (Part (b)), to prove our claim we need to show that during I_1 $\mathcal{K}$ does not cease to be a releaser and no process becomes a releaser. Suppose not, i.e., the claim $R(t) = \{\mathcal{K}\}$ fails to hold for the first time at some point t' in I_1.

Case a - Process $\mathcal{K}$ ceases to be a releaser of L at t': By definition, cease-release event $\phi_{\mathcal{K}}$ occurs when $\mathcal{K}$ executes a successful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36, From Part (e), $\mathcal{K}$ executes a $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36. If $\mathcal{K}$ executes a successful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36 then, by definition, cease-release event $\phi_{\mathcal{K}}$ occurs and by Claim 4.4.12 $\mathcal{K}$ ceases to be the releaser of L. Thus, $t' = t_{\mathcal{K}}^{36}$ and Ctr changes to value of 0 at t'. But since $t' \in I_1$ and $\mathsf{Ctr} = 1$ throughout I_1, we have a contradiction. If $\mathcal{K}$ executes an unsuccessful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36, then $\mathsf{Ctr} = 1$ at $t_{\mathcal{K}}^{36-}$. Since p did not cease to a releaser at $t_{\mathcal{K}}^{36-}$, $t_{\mathcal{K}}^{36-} < t'$. Since $I_1^- = t_{\mathcal{K}}^{5} < t_{\mathcal{K}}^{36} < t' < I_1^+$ and $\mathsf{Ctr} = 1$ throughout I_1, $\mathsf{Ctr} = 1$ at $t_{\mathcal{K}}^{36-}$, and thus we have a contradiction.

Case b - Some process q becomes a releaser of L at t': Since Ctr is not increased during I_1, it follows from conditions (R1) and (R2) that some process r promoted q at time t'. Then by definition, event π_r occurs at t', and thus from Claim 4.4.12 it follows that r is a releaser of L immediately before t'. Since $\mathcal{K}$ is the only releaser immediately before t', $r = \mathcal{K}$. Then cease-release event $\pi_{\mathcal{K}}$ occurred at t' and $\mathcal{K}$ ceases to be a releaser at t'. As was shown in **Case a**, this leads to a contradiction.

Proof of (g): At time I_1^- the claim $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty holds by Part (b). Suppose some process p changes $\mathsf{Sync2}$ or $\mathsf{Sync1}$ or $\mathsf{PawnSet}$ for the first time at some point t' during I_1. From Claim 4.4.13 it follows that p is a releaser of lock L at time t'. Since for all $t \in I_1$, $R(t) = \{\mathcal{K}\}$ (Part (f)), it follows that $p = \mathcal{K}$. From Part (e), $\mathcal{K}$ does not change any of the variables before the point when it executes a $\mathsf{Ctr}.\mathtt{CAS}(1, 0)$ operation in line 36, i.e., $t_{\mathcal{K}}^{36-} < t'$. If $\mathcal{K}$ executes a successful $\mathsf{Ctr}.\mathtt{CAS}(1, 0)$ operation in line 36 then the interval I_1 ends and clearly $t' \notin I_1$, hence a contradiction. If $\mathcal{K}$ executes an unsuccessful $\mathsf{Ctr}.\mathtt{CAS}(1, 0)$ operation in line 36 then $\mathsf{Ctr} = 1$ at $t_{\mathcal{K}}^{36-}$. Since $I_1^- = t_{\mathcal{K}}^{5-} < t_{\mathcal{K}}^{36-} < t' < I_1^+$ and $\mathsf{Ctr} = 1$ throughout I_1, we have a contradiction. $\square$

Claim 4.4.22. *If $I_2 = \varnothing$ and $R(I_0^-) = \varnothing$ and at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty, then the following claims hold:*

(a) $R(I_2^-) = \{\mathcal{K}, \mathcal{Q}\}$ and at time I_2^-, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty.

(b) $\mathcal{K}$ and $\mathcal{Q}$ are the first two releasers of L.

(c) During $(I_2^-, I_2^+]$ a process can become a releaser of L only if it gets promoted by a releaser of L.

(d) If $\mathcal{K}$ takes enough steps, $\mathcal{K}$ executes lines of code of $\mathtt{release}_{\mathcal{K}}\mathtt{()}$ starting with line 34 as depicted in Figure 4.9.

(e) *$\mathcal{K}$ executes an unsuccessful* `Ctr.CAS(1,0)` *operation in line 36, and calls* `helpRelease`$_{\mathcal{K}}$`()` *in line 39 such that $I_2^- < t_{\mathcal{K}}^{36-} < t_{\mathcal{K}}^{39-}$.*

(f) *If $\mathcal{K}$ and $\mathcal{Q}$ take enough steps, $\mathcal{Q}$ finishes* `lock`$_{\mathcal{Q}}$`()` *during T.*

(g) *If $\mathcal{K}$ and $\mathcal{Q}$ take enough steps, $\mathcal{Q}$ executes lines of code of* `lock`$_{\mathcal{Q}}$`()` *starting with line 2 as depicted in Figure 4.10.*

(h) *If $\mathcal{Q}$ calls* `release`$_{\mathcal{Q}}$`()`*, it executes lines of code of* `release`$_{\mathcal{Q}}$`()` *starting with line 34 as depicted in Figure 4.11.*

(i) *$\mathcal{Q}$ calls* `helpRelease`$_{\mathcal{Q}}$`()` *either in line 30 or in line 43, after time I_2^-.*

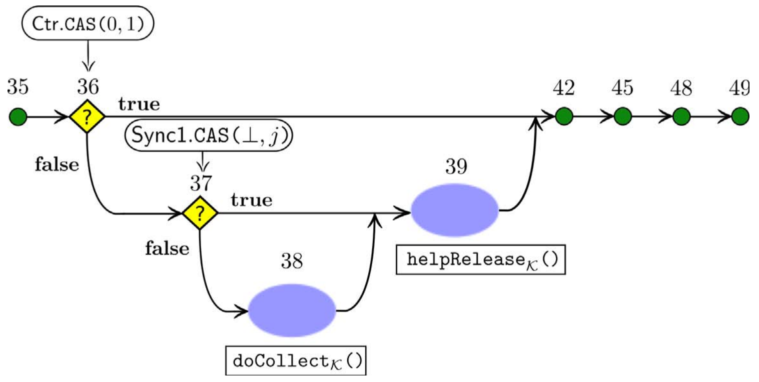


Figure 4.9: $\mathcal{K}$'s call to `release`$_{\mathcal{K}}$`(`j`)`

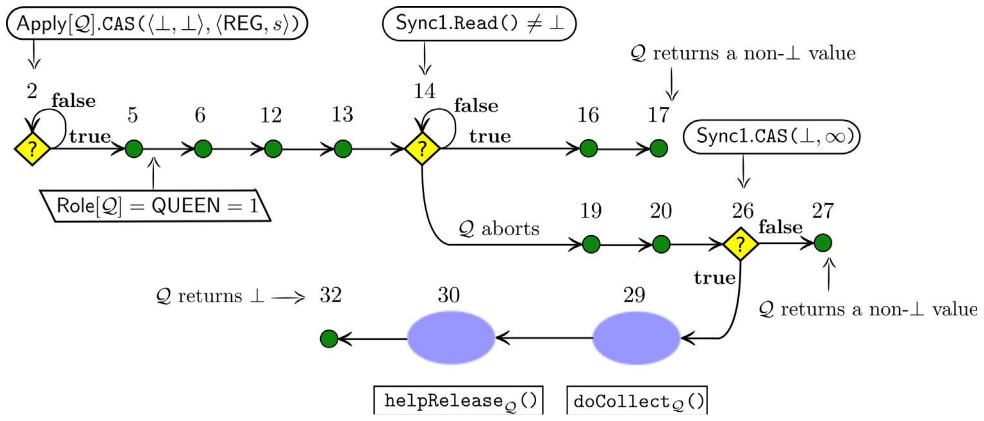


Figure 4.10: $\mathcal{Q}$'s call to `lock`$_{\mathcal{Q}}$`()`

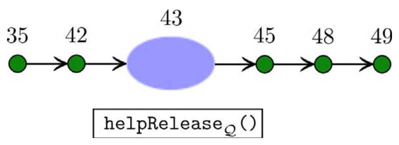


Figure 4.11: $\mathcal{Q}$'s call to $\texttt{release}_{\mathcal{Q}}\texttt{()}$

Proof. **Proof of (a) and (b):** Since $\mathcal{Q}$ is the process that increases Ctr from 1 to 2 at time I_2^-, and since $\mathcal{Q}$ can increase Ctr only by executing a $\mathsf{Ctr}.\texttt{inc()}$ operation in line 5 (by Claim 4.4.19) $\mathcal{Q}$ becomes a releaser of lock L by condition (R1) at $I_2^- = t_{\mathcal{Q}}^5$. Since for all $t \in I_1$, $R(t) = \{K\}$ (Claim 4.4.21(f)), it follows that $R(I_2^-) = \{\mathcal{K}, \mathcal{Q}\}$. By claim 4.4.21(g), throughout I_1, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty, and since the only operation executed at time I_2^- is $\mathsf{Ctr}.\texttt{inc()}$, it follows that at time I_2^-, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty. Hence Part (b) holds. Clearly $\mathcal{K}$ and $\mathcal{Q}$ are the first two releasers of L, hence Part (b) holds.

Proof of (c): From conditions (R1) and (R2), a process can become a releaser of L either by increasing Ctr to 1 or 2 or by getting promoted. Since Ctr is not increased during $(I_2^-, I_2^+]$, it follows that during $(I_2^-, I_2^+]$ a process becomes a releaser of L only if it gets promoted. By definition, a process can be promoted only when a $\mathsf{PawnSet}.\texttt{promote()}$ operation is executed in line 65 and from Claim 4.4.16 only a releaser of L can execute this operation. Then during $(I_2^-, I_2^+]$ a process becomes a releaser of L only if it gets promoted by a releaser of L.

Proof of (d) and (e): From Claim 4.4.21(e), $\mathcal{K}$ executes the $\mathsf{Ctr}.\texttt{CAS}(1, 0)$ operation in line 36 during T. If $\mathcal{K}$'s $\mathsf{Ctr}.\texttt{CAS}(1, 0)$ operation is successful then the value of Ctr decreases from 1 to 0 and the Ctr-cycle interval T ends and thus $I_2 = \varnothing$, which is a contradiction to our assumption that $I_2 = \varnothing$. Then $\mathcal{K}$'s $\mathsf{Ctr}.\texttt{CAS}(1, 0)$ operation is unsuccessful.

Since $\mathcal{K}$ executes an unsuccessful $\mathsf{Ctr}.\texttt{CAS}(1, 0)$ operation in line 36, $\mathcal{K}$ satisfies the if-condition of line 36, executes lines 37-38 and calls $\texttt{helpRelease}_{\mathcal{K}}\texttt{()}$ in line 39, and then executes lines 49-50.

Since $\mathcal{K}$ executes an unsuccessful $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation in line 36, it follows that Ctr was changed from 1 to 2 at time I_2^- (by definition), and thus $I_2^- < t_{\mathcal{K}}^{36}$. Since $t_{\mathcal{K}}^{36} < t_{\mathcal{K}}^{39}$, it follows that $I_2^- < t_{\mathcal{K}}^{36} < t_{\mathcal{K}}^{39}$.

Proof of (f), (g) and (h): Since $\mathcal{Q}$ is the process that increases Ctr from 1 to 2 at time I_2^-, and since $\mathcal{Q}$ can increase Ctr only by executing a $\mathsf{Ctr}.\mathtt{inc()}$ operation in line 5 (by Claim 4.4.19) $\mathcal{Q}$ set $\mathsf{Role}[\mathcal{K}] = 1 = \mathsf{QUEEN}$ at $t_{\mathcal{Q}}^{5} = I_2^-$. Then from the code structure, $\mathcal{Q}$ does not execute lines 7-9, and does not repeat the role-loop, instead, it proceeds to line 13 and then proceeds to busy-wait in the spin loop of line 14. Then $\mathcal{Q}$ does not finish $\mathtt{lock}_{\mathcal{Q}}\mathtt{()}$ only if it spins indefinitely in line 14 and does not receive a signal to abort.

For the purpose of a contradiction assume that $\mathcal{Q}$ does not finish $\mathtt{lock}_{\mathcal{Q}}\mathtt{()}$. Then $\mathcal{Q}$ reads the value $\perp$ from $\mathsf{Sync1}$ in line 14 indefinitely. From Part (e) it follows that $\mathcal{K}$ executes a $\mathsf{Sync1}.\mathtt{CAS}(\perp, j)$ operation in line 37 during $(I_2^-, I_2^+]$. Since $\mathsf{Sync1} = \perp$ at time I_2^- (Part (a)), and only a releaser can change $\mathsf{Sync1}$ (Claim 4.4.13), and $\mathcal{Q}$ is busy-waiting in line 14, it follows that the only other releaser, $\mathcal{K}$, executed a successful $\mathsf{Sync1}.\mathtt{CAS}(\perp, j)$ operation in line 37 during $(I_2^-, I_2^+]$ and changed $\mathsf{Sync1}$ to a non-$\perp$ value. Then for $\mathcal{Q}$ to read $\perp$ from $\mathsf{Sync1}$ in line 14 indefinitely, some process must reset $\mathsf{Sync1}$ to $\perp$ before $\mathcal{Q}$ reads $\mathsf{Sync1}$ again.

Case a - $\mathcal{K}$ resets $\mathsf{Sync1}$ in line 58 before $\mathcal{Q}$ reads $\mathsf{Sync1}$ again: For $\mathcal{K}$ to reset $\mathsf{Sync1}$ in line 58, $\mathcal{K}$ must satisfy the if-condition of line 56 and thus $\mathcal{K}$ must execute an unsuccessful $\mathsf{Sync2}.\mathtt{CAS}(\perp, \mathcal{K})$ operation in line 56. Since $\mathsf{Sync2} = \perp$ at time I_2^- (Part (a)), and only a releaser can change $\mathsf{Sync2}$ (Claim 4.4.13), and $\mathcal{Q}$ is busy-waiting in line 14, it follows that $\mathsf{Sync2} = \perp$ at $t_{\mathcal{K}}^{56-}$. Thus $\mathcal{K}$'s $\mathsf{Sync2}.\mathtt{CAS}(\perp, \mathcal{K})$ operation in line 56 is successful and we get a contradiction.

Case b - some other process becomes a releaser and resets $\mathsf{Sync1}$ before $\mathcal{Q}$ reads

Sync1 again: From Part (c) it follows that during $(I_2^-, I_2^+]$ a process can become a releaser of L only if it is promoted (by condition (R2)). Since a process is promoted only by a releaser of L and $\mathcal{K}$ is the only other releaser of L apart from $\mathcal{Q}$, it follows that $\mathcal{K}$ promotes some process before $\mathcal{Q}$ reads Sync1 again. As argued in **Case a**, $\mathcal{K}$ executes a successful `Sync2.CAS(`$\perp, \mathcal{K}$`)` operation in line 56. Then from the code structure, $\mathcal{K}$ does not call `doPromote`$_\mathcal{K}$`()` in line 62, and thus $\mathcal{K}$ does not promote any process. Hence, we have a contradiction.

Proof of (i): Since $\mathcal{Q}$ is the process that increases Ctr from 1 to 2 at time I_2^-, and since $\mathcal{Q}$ can increase Ctr only by executing a `Ctr.inc()` operation in line 5 (by Claim 4.4.19) $\mathcal{Q}$ set $\mathsf{Role}[\mathcal{K}] = 1 = \mathsf{QUEEN}$ at $t_\mathcal{Q}^5$. Then from the code structure, $\mathcal{Q}$ does not execute lines 7-9, and does not repeat the role-loopp; instead, it proceeds to line 13 and then proceeds to busy-wait in the spin loop of line 14.

Case a - $\mathcal{Q}$ does not receive a signal to abort while busy-waiting in line 14: From Part (f), $\mathcal{Q}$ does not busy-wait indefinitely in line 14 and eventually breaks out. Since $\mathcal{Q}$ breaks out of the spin loop of line 14 it reads non-$\perp$ from Sync1 and then from the code structure it follows that $\mathcal{Q}$ goes on to return that non-$\perp$ value in line 17. Consequently $\mathcal{Q}$ calls `release`$_\mathcal{Q}$`(`j`)` (follows from conditions b and c). Consider $\mathcal{Q}$'s call to `release`$_\mathcal{Q}$`(`j`)`. Since $\mathcal{Q}$ last changed $\mathsf{Role}[\mathcal{Q}]$ only in line 5, $\mathsf{Role}[\mathcal{Q}] = \mathsf{QUEEN}$ at $t_\mathcal{Q}^{34-}$. Since $\mathsf{Role}[\mathcal{Q}]$ is unchanged during `release`$_\mathcal{Q}$`()` (Claim 4.4.2(b)), it follows that $\mathsf{Role}[\mathcal{Q}] = \mathsf{QUEEN}$ throughout `release`$_\mathcal{Q}$`()`. Then from the code structure it follows that $\mathcal{Q}$ executes only lines 34-35, 42-45 and 49-50. Then $\mathcal{Q}$ calls `helpRelease`$_\mathcal{Q}$`()` only in line 43, and since $I_2^- = t_\mathcal{Q}^5 < t_\mathcal{Q}^{43}$, our claim holds.

Case b - $\mathcal{Q}$ receives a signal to abort while busy-waiting in line 14: Then $\mathcal{Q}$ calls `abort`$_\mathcal{Q}$`()`, and from the code structure $\mathcal{Q}$ executes lines 18-20, and then line 26. If $\mathcal{Q}$ fails the `Sync1.CAS(`$\perp, \infty$`)` operation of line 26, then $\mathsf{Sync1} = \perp$ at $t_\mathcal{Q}^{26-}$. From Claim 4.4.13, only a releasers of L can change Sync1 to a non-$\perp$ value, and since $\mathcal{K}$ and $\mathcal{Q}$ are the only releasers of L, it follows that $\mathcal{K}$ changed Sync1 to a non-$\perp$ value. Then $\mathcal{Q}$ satisfies the if-condition of line 26 and returns the non-$\perp$ value written by $\mathcal{K}$ to Sync1 in line 27. Consequently $\mathcal{Q}$ calls `release`$_{\mathcal{Q}}$`(`j`)` (follows from conditions b and c), and as argued in **Case a**, $\mathcal{Q}$ executes only lines 34-35, 42-45 and 49-50, and $\mathcal{Q}$ calls `helpRelease`$_{\mathcal{Q}}$`()` only in line 43. Since $I_2^- = t_{\mathcal{Q}}^5 < t_{\mathcal{Q}}^{43}$, our claim holds.

If $\mathcal{Q}$'s Sync1.`CAS(`$\perp, \infty$`)` operation is successful, then $\mathcal{Q}$ goes on to call `doCollect`$_{\mathcal{Q}}$`()` in line 29, calls `helpRelease`$_{\mathcal{Q}}$`()` in line 30, then executes lines 32-33, and finally returns $\perp$ in line 33. Since $I_2^- = t_{\mathcal{Q}}^5 < t_{\mathcal{Q}}^{30}$, our claim holds. □

Define λ to be the first point in time when Sync2 is changed to a non-$\perp$ value, and if Sync2 is never changed to non-$\perp$ then $\lambda = \infty$. Define γ to be the first point in time when a PawnSet.`promote()` operation is executed, and if a PawnSet.`promote()` operation is never executed then $\gamma = \infty$. From Claims 4.4.22(e) and 4.4.22(i), both $\mathcal{K}$ and $\mathcal{Q}$ execute `helpRelease`$_{\mathcal{K}}$`()` and `helpRelease`$_{\mathcal{Q}}$`()`, respectively, after time I_2^-. Let $\mathcal{A} \in \{\mathcal{K}, \mathcal{Q}\}$ be the first process among them to execute line 56, and let $\mathcal{B} \in \{\mathcal{K}, \mathcal{Q}\} - \{\mathcal{A}\}$ be the other process, i.e., $t_{\mathcal{A}}^{56} < t_{\mathcal{B}}^{56}$.

Claim 4.4.23. *If $I_2 = \varnothing$ and $R(I_0^-) = \varnothing$ and at I_0^-,* Sync1 $=$ Sync2 $= \perp$ *and* PawnSet *is candidate-empty, then the following claims hold:*

(a) $I_2^- < \lambda = t_{\mathcal{A}}^{56}$ and for all $t \in [I_2^-, \lambda)$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$ and Sync2 $= \perp$ *throughout $[I_2^-, \lambda)$, and cease-release event $\tau_{\mathcal{A}}$ occurs at λ.*

(b) If $\mathcal{K}$ and $\mathcal{Q}$ take enough steps, then $\mathcal{A}$ executes lines of code of `helpRelease`$_{\mathcal{A}}$`()` *starting with line 56 as depicted in Figure 4.12.*

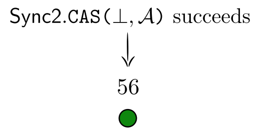


Figure 4.12: $\mathcal{A}$'s call to `helpRelease`$_{\mathcal{A}}$`()`

(c) If $\mathcal{K}$ and $\mathcal{Q}$ take enough steps, then $\mathcal{B}$ executes lines of code of `helpRelease`$_\mathcal{B}$`()` *and* `doPromote`$_\mathcal{B}$`()` *as depicted in Figures 4.13 and 4.14, respectively.*

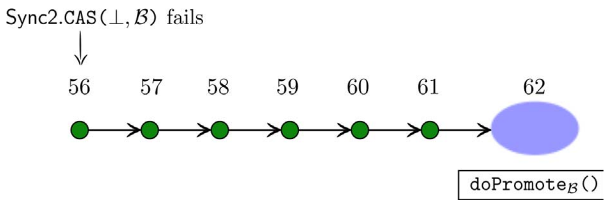


Figure 4.13: $\mathcal{B}$'s call to `helpRelease`$_\mathcal{B}$`()`

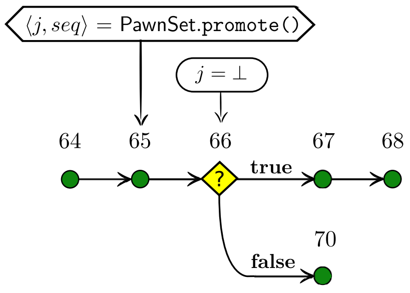


Figure 4.14: $\mathcal{B}$'s call to `doPromote`$_\mathcal{B}$`()`

(d) $\lambda < \gamma = t_\mathcal{B}^{65}$.

(e) $\forall_{t\in[\lambda,\gamma)}$, $R(t) = \{\mathcal{B}\}$.

(f) At time γ, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$.

(g) No promotion event occurs at lock L *during $[I_2^-, \gamma)$.*

(h) The PawnSet.promote() *operation at time γ does not return a value in* $\{\langle a,b\rangle | a \in \{\mathcal{K},\mathcal{Q}\}, b \in \mathbb{N}\}$.

(i) If the PawnSet.promote() *operation at time γ returns a non-$\langle\perp,\perp\rangle$ value then $\mathcal{B}$'s cease-release event $\pi_\mathcal{B}$ occurs at time γ.*

(j) If the PawnSet.promote() *operation at time γ returns value $\langle\perp,\perp\rangle$ then $\mathcal{B}$'s cease-release event $\theta_\mathcal{B}$ occurs at $t' = t_\mathcal{B}^{68} \geq \gamma$, and throughout $[\gamma, t']$ no process is promoted, and $\forall_{t\in[\gamma,t')}$, $R(t) = \{\mathcal{B}\}$.*

(k) Either $\mathcal{K}$ or $\mathcal{Q}$ calls `doCollect()`*, specifically during* $[I_2^-, \gamma]$.

Proof. **Proof of (a):** We first show that for all $t \in [I_2^-, t_{\mathcal{A}}^{56-}]$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$ and then show that $\lambda = t_{\mathcal{A}}^{56}$. From Claims 4.4.22(e) $\mathcal{K}$ calls $\texttt{helpRelease}_{\mathcal{K}}\texttt{()}$ in line 39 after time I_2^-. From Claim 4.4.22(a), $\mathcal{K}$ is a releaser of L at time I_2^-. From an inspection of Figures 4.8 and 4.9, throughout $[I_1, t_K^{39-}]$ $\mathcal{K}$ does not execute a call to $\texttt{helpRelease}_{\mathcal{K}}\texttt{()}$ or $\texttt{doPromote}_{\mathcal{K}}\texttt{()}$. Also from an inspection, $\mathcal{K}$ fails to decrease Ctr from 1 to 0 at $t_{\mathcal{K}}^{36}$, thus $\mathcal{K}$'s cease-release event $\phi_{\mathcal{K}}$ does not occur. Since $\mathcal{K}$'s cease-release events $\tau_{\mathcal{K}}, \pi_{\mathcal{K}}$ and $\theta_{\mathcal{K}}$ only occur during $\texttt{helpRelease}_{\mathcal{K}}\texttt{()}$ or $\texttt{doPromote}_{\mathcal{K}}\texttt{()}$ (Claims 4.4.5(e) and 4.4.5(f)), it follows that $\mathcal{K}$ is a releaser of L throughout $[I_1^-, t_{\mathcal{K}}^{56-}]$.

From Claim 4.4.22(i), $\mathcal{Q}$ calls $\texttt{helpRelease}_{\mathcal{Q}}\texttt{()}$, respectively either in line 30 or line 43, after time I_2^-. From Claim 4.4.22(a), $\mathcal{Q}$ is a releaser of L at time I_2^-. From an inspection of Figures 4.10 and 4.11, throughout $[I_2, t_Q^{56-}]$ $\mathcal{Q}$ does not execute a call to $\texttt{helpRelease}_{\mathcal{Q}}\texttt{()}$ or $\texttt{doPromote}_{\mathcal{Q}}\texttt{()}$. Also from an inspection, $\mathcal{Q}$ does not execute a $\mathsf{Ctr}.\texttt{CAS}(1, 0)$ operation in line 36, and thus $\mathcal{Q}$'s cease release event $\phi_{\mathcal{Q}}$ does not occur. Since $\mathcal{Q}$'s cease-release events $\tau_{\mathcal{Q}}, \pi_{\mathcal{Q}}$ and $\theta_{\mathcal{Q}}$ only occur during $\texttt{helpRelease}_{\mathcal{Q}}\texttt{()}$ or $\texttt{doPromote}_{\mathcal{Q}}\texttt{()}$ (Claims 4.4.5(e) and 4.4.5(f)), it follows that $\mathcal{Q}$ is a releaser of L throughout $[I_2^-, t_{\mathcal{Q}}^{56-}]$.

Then for all $t \in [I_2^-, t_{\mathcal{A}}^{56-}]$, $\{\mathcal{K}, \mathcal{Q}\} \subseteq R(t)$ since $I_1^- < I_2^-$ and $t_{\mathcal{A}}^{56-} = \texttt{min}(t_{\mathcal{K}}^{56-},\ t_{\mathcal{Q}}^{56-}\texttt{)}$. From Claim 4.4.22(c), it follows that a process can become a releaser during I_2 only if it is promoted by a releaser of L. Then to show that for all $t \in [I_2^-, t_{\mathcal{A}}^{56-}]$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$, we need to show that no process is promoted by $\mathcal{K}$ or $\mathcal{Q}$ during $[I_2^-, t_{\mathcal{A}}^{56-}]$. If a process was promoted by $\mathcal{K}$ or $\mathcal{Q}$ during $[I_2^-, t_{\mathcal{A}}^{56-}]$ then by definition cease-release events $\pi_{\mathcal{K}}$ or $\pi_{\mathcal{Q}}$ would have occurred during $[I_2^-, t_{\mathcal{A}}^{56-}]$, but as shown above this does not happen.

From Claim 4.4.22(a), $\mathsf{Sync2} = \bot$ at time I_2^-. From a code inspection, $\mathsf{Sync2}$ is changed to a non-$\bot$ value only in line 56 (during `helpRelease()`), moreover only by a releaser of L (from Claim 4.4.13). Since for all $t \in [I_2^-, t_{\mathcal{A}}^{56-}]$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$ and

$t_{\mathcal{A}}^{56-} = \texttt{min}(t_{\mathcal{K}}^{56-}, t_{\mathcal{Q}}^{56-})$, it follows then that $\mathsf{Sync2} = \perp$ throughout $[I_2^-, t_{\mathcal{A}}^{56-}]$ and $\mathcal{A}$ executes a successful $\mathsf{Sync2}.\texttt{CAS}(\perp, \mathcal{A})$ operation in line 56. Thus $\mathcal{A}$'s cease-release event $\tau_{\mathcal{A}}$ occurs at $t_{\mathcal{A}}^{56}$.

Since $\mathsf{Sync2} = \perp$ throughout $[I_0^-, I_1^+]$ (Claims 4.4.21(a) and 4.4.21(g)) and throughout $[I_2^-, t_{\mathcal{A}}^{56-}]$, it follows that $\mathsf{Sync2}$ was changed to a non-$\perp$ value for the first time at $t_{\mathcal{A}}^{56}$, thus $\lambda = t_{\mathcal{A}}^{56}$. Then it follows for all $t \in [I_2^-, \lambda)$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$, and $\mathsf{Sync2} = \perp$ throughout $[I_2^-, \lambda)$

Proof of (b): From Part (a), $\mathcal{A}$'s cease-release event τ_A occurs at $\lambda = t_A^{56}$, and thus $\mathcal{A}$'s $\mathsf{Sync2}.\texttt{CAS}(\perp, \mathcal{A})$ operation in line 56 succeeds. Then from the code structure $\mathcal{A}$ does not satisfy the if-condition on line 56 and returns from its call to $\texttt{helpRelease}_{\mathcal{A}}\texttt{()}$. Thus, Figure 4.12 follows.

Proof of (c), (d), (e), (f), (g), (h), (i) and (j): From Part (a), $\lambda = t_{\mathcal{A}}^{56}$ and for all $t \in [I_2^-, \lambda)$, $R(t) = \{\mathcal{K}, Q\}$ and $\mathsf{Sync2} = \perp$ throughout $[I_2^-, \lambda)$ and cease-release event $\tau_{\mathcal{A}}$ occurs at λ. Then $\mathcal{A}$ ceases to be a releaser of L at λ, and thus $R(\lambda) = \{\mathcal{B}\}$ and $\mathsf{Sync2} = \mathcal{A} = \perp$ at λ. From Claim 4.4.22(c) it follows that $\mathcal{B}$ will continue to be the only releaser of L until the point when $\mathcal{B}$ ceases to be a releaser of L or promotes another process. Let $t > \lambda$ be the point in time when $\mathcal{B}$ ceases to be a releaser of L. Since $\mathcal{B}$ ceases to be a releaser of L if it promotes another process (by definition of cease-release event $\pi_{\mathcal{B}}$), it follows that $\mathcal{B}$ is the only releaser of L throughout $[\lambda, t)$. Then from Claim 4.4.13 it follows that $\mathcal{B}$ has exclusive write-access to $\mathsf{Sync1}, \mathsf{Sync2}$ and exclusive registration-access to $\mathsf{PawnSet}$ throughout $[\lambda, t)$.

Now consider $\mathcal{B}$'s $\texttt{helpRelease}_{\mathcal{B}}\texttt{()}$ call. Since $\lambda = t_{\mathcal{A}}^{56} < t_{\mathcal{B}}^{56}$ and $\mathsf{Sync2} = \perp$ at λ and $\mathcal{B}$ has exclusive write-access to $\mathsf{Sync2}$ throughout $[\lambda, t)$, $\mathcal{B}$ fails the $\mathsf{Sync2}.\texttt{CAS}(\perp, \mathcal{B})$ operation at $t_{\mathcal{B}}^{56}$, and thus satisfies the if-condition of line 56. It then executes lines 57 - 62, and calls $\texttt{doPromote}_{\mathcal{B}}\texttt{()}$ in line 62. Then Figures 4.13 and 4.14 and Part (c) follows immediately.

We now show that $\gamma = t^{65}_{\mathcal{B}} \leq t$. Since $\lambda = t^{56}_{\mathcal{A}}$ and $t^{56}_{\mathcal{A}} < t^{56}_{\mathcal{B}} < t^{65}_{\mathcal{B}}$, it would follow that $\lambda < \gamma$, and hence we would have proved Part (d). And since $\mathcal{B}$ is the only releaser of L throughout $[\lambda, t)$, we would have proved Part (e) as well, i.e., $\mathcal{B}$ is the only releaser throughout $[\lambda, \gamma)$.

During $\texttt{doPromote}_{\mathcal{B}}\texttt{()}$, $\mathcal{B}$ executes a $\mathsf{PawnSet}.\texttt{promote()}$ operation in line 65. Since $\mathcal{K}$ and $\mathcal{Q}$ are the first two releasers of L during T (Claim 4.4.22(b)), and only a releaser executes a $\mathsf{PawnSet}.\texttt{promote()}$ operation (Claim 4.4.16), and $\mathcal{A}$ ceased to be a releaser at $t^{56}_{\mathcal{A}} < t^{65}_{\mathcal{B}}$, it follows that $\mathcal{B}$'s $\mathsf{PawnSet}.\texttt{promote()}$ operation in line 65 is the first $\mathsf{PawnSet}.\texttt{promote()}$ operation, and thus $\gamma = t^{65}_{\mathcal{B}}$. Since none of $\mathcal{B}$'s cease-release events occur during $[t^{56}_{\mathcal{B}}, t^{65-}_{\mathcal{B}}]$, $t \geq t^{65}_{\mathcal{B}}$.

During $[t^{56}_{\mathcal{B}}, t^{65}_{\mathcal{B}}]$, $\mathcal{B}$ resets $\mathsf{Sync1}$ and $\mathsf{Sync2}$ in lines 58 and 60, respectively, and since $\mathcal{B}$ has exclusive write-access to $\mathsf{Sync1}$ and $\mathsf{Sync2}$ throughout $[t^{56}_{\mathcal{B}}, t^{65}_{\mathcal{B}}]$, at time $\gamma = t^{65}_{\mathcal{B}}$, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$. Thus, Part (f) follows.

By definition γ is the point in time when the first $\mathsf{PawnSet}.\texttt{promote()}$ operation occurs. Since a promotion event occurs only when a $\mathsf{PawnSet}.\texttt{promote()}$ operation returns a non-$\langle\perp,\perp\rangle$ value, it follows that no promotion event occurs during $[I_0^-, \gamma)$. Hence, Part (g) follows.

Since $\mathcal{B}$ has exclusive write-access to $\mathsf{Sync2}$ throughout $[\lambda, t^{65}_{\mathcal{B}}]$, and $\mathsf{Sync2} = \mathcal{A}$ at $\lambda > t^{56}_{\mathcal{B}}$, $\mathcal{B}$ reads the value $\mathcal{A}$ from $\mathsf{Sync2}$ in line 59 and executes a $\mathsf{PawnSet}.\texttt{remove(}\mathcal{A}\texttt{)}$ operation in line 61. Since $\mathcal{B}$ executes $\mathsf{PawnSet}.\texttt{remove(}\mathcal{A}\texttt{)}$ and $\mathsf{PawnSet}.\texttt{remove(}\mathcal{B}\texttt{)}$ in lines 61 and 64 during $[\lambda, \gamma)$ and $\mathcal{B}$ has exclusive registration-access to $\mathsf{PawnSet}$ during $[\lambda, \gamma)$, it follows from the semantics of the $\mathsf{AbortableProArray}_n$ object that $\mathcal{B}$'s $\mathsf{PawnSet}.\texttt{promote()}$ operation at time γ does not return values in $\{\langle a, b\rangle | a \in \{\mathcal{A}, \mathcal{B}\} = \{\mathcal{K}, \mathcal{Q}\}, b \in \mathbb{N}\}$. Hence, Part (h) follows.

Case a - $\mathcal{B}$'s $\mathsf{PawnSet}.\texttt{promote()}$ operation returns a non-$\langle\perp,\perp\rangle$ value: Then $\mathcal{B}$'s cease-release event $\pi_{\mathcal{B}}$ occurs at $t^{65}_{\mathcal{B}} = \gamma$ (Claim 4.4.16), and thus Part (i) holds.

Case b - $\mathcal{B}$'s PawnSet.`promote()` operation in line 65 returns $\langle\bot,\bot\rangle$. Then $\mathcal{B}$ did not find any process to promote, and thus cease-release event $\pi_{\mathcal{B}}$ did not occur. From the code structure $\mathcal{B}$ goes on to execute a Ctr.`CAS`$(2,0)$ operation in line 68. Since Ctr $= 2$ throughout I_2, it follows that $\mathcal{B}$'s Ctr.`CAS`$(2,0)$ operation succeeds, and thus $\mathcal{B}$'s cease-release event $\theta_{\mathcal{B}}$ occurs at $t^{68}_{\mathcal{B}}$ and the intervals I_2 and T end. Therefore $t' = t^{68}_{\mathcal{B}} > t^{65}_{\mathcal{B}} = \gamma$. Clearly, $\mathcal{B}$ does not promote any process in $[t^{65}_{\mathcal{B}}, t^{68}_{\mathcal{B}}] = [\gamma, t']$, and thus Part (j) holds.

Proof of (k): From an inspection of Figure 4.9, $\mathcal{K}$ executes a Sync1.`CAS`$(\bot,\mathcal{K})$ operation in line 37. Since $I_2^- < t^{36}_{\mathcal{K}} < t^{37}_{\mathcal{K}} < t^{56}_{\mathcal{K}} < \gamma$ (from Parts (a) and (d)), it follows that $t^{37}_{\mathcal{K}} \in [I_2^-, \gamma]$. From an inspection of Figures 4.10 and 4.10, $\mathcal{Q}$ may or may not execute a Sync1.`CAS`$(\bot,\infty)$ operation in line 26. If $\mathcal{Q}$ executes a Sync1.`CAS`$(\bot,\infty)$ operation in line 26, since $I_2^- < t^{26}_{\mathcal{Q}} < t^{56}_{\mathcal{Q}} < \gamma$, it follows that $t^{26}_{\mathcal{Q}} \in [I_2^-, \gamma]$.

Since for all $t \in [I_2^-, \gamma]$, $R(t) \subseteq \{\mathcal{K}, \mathcal{Q}\}$ (from Parts (a) and (e)), and only releasers of L have write-access to Sync1 (Claim 4.4.13), and Sync1 $= \bot$ at I_2^- (Claim 4.4.22(a)), it follows that either $\mathcal{K}$ or $\mathcal{Q}$ executes a successful `CAS()` operation on Sync1. Then from the code structure it follows that either $\mathcal{K}$ or $\mathcal{Q}$ executed a call to `doCollect()` in lines 38 or 29, respectively. Since $t^{38}_{\mathcal{K}} < t^{39-}_{\mathcal{K}} = t^{56-}_{\mathcal{K}} < \gamma$ and $t^{29}_{\mathcal{Q}} < t^{56-}_{\mathcal{Q}} < \gamma$, $\mathcal{K}$ or $\mathcal{Q}$ executed a call to `doCollect()` during $[I_2^-, \gamma]$. □

Claim 4.4.24. *If a process p is promoted at time $t' \in T$ and a* PawnSet.`reset()` *has not been executed during $[I_0^-, t']$, then p did not execute a* PawnSet.`abort`(p,s) *operation during $[I_0^-, t']$, where $s \in \mathbb{N}$.*

Proof. Suppose not, i.e., p executed a PawnSet.`abort`(p,s) operation at time $t < t'$. Since p has not been promoted before $t' > t$ it follows that a PawnSet.`promote()` operation that returns $\langle p, \cdot\rangle$ has not been executed before t. Then from Claim 4.4.3(a) and the semantics of PawnSet, it follows that the p-th entry of PawnSet is not at value $\langle \mathsf{PRO}, s\rangle$ $= \langle 2, s\rangle$ throughout $[I_0^-, t]$. Then p's PawnSet.`abort`(p,s) operation at t succeeds, and thus p writes value $\langle \mathsf{ABORT}, s\rangle = \langle 3, s\rangle$ to the p-entry of PawnSet. Then for p to be promoted at $t' > t$, it follows from the semantics of PawnSet and Claim 4.4.3(a), that during $[t, t')$ a `PawnSet.reset()` operation and then a `PawnSet.collect(`A`)` operation where $A[p] = s$, must be executed, followed by a `PawnSet.promote()` at t' that returns $\langle p, s\rangle$. This is a contradiction to the assumption that a `PawnSet.reset()` is not executed during $[I_0, t']$. $\square$

Let ℓ be the number of times a promotion occurs during T. For all $i \in \{1, \ldots, \ell\}$, define Ω_i to be the i-th interval $[\Omega_i^-, \Omega_i^+]$ that begins when the i-th promotion occurs during T and ends when the promoted process ceases to be a releaser of L. Let $\mathcal{P}_i$ be the process promoted at Ω_i^-.

Claim 4.4.25. *If $I_2 = \varnothing$ and $R(I_0^-) = \varnothing$ and at time I_0^-, Sync1 = Sync2 = $\perp$ and PawnSet is candidate-empty, then the following claims hold for all $i \in \{1, \ldots, \ell\}$:*

(a) If $\ell \geq 1$, then $\gamma = \Omega_1^-$ and $R(\Omega_1^-) = \{\mathcal{P}_1\}$, and Sync1 = Sync2 = $\perp$ at Ω_1^-, and no `PawnSet.reset()` operation has been executed during $[I_0^-, \Omega_1^-]$.

(b) If $R(\Omega_i^-) = \{\mathcal{P}_i\}$, then for all $t \in [\Omega_i^-, \Omega_i^+)$, $R(t) = \{\mathcal{P}_i\}$. (i.e., $\mathcal{P}_i$ is the only releaser throughout Ω_i)

(c) If $i = \ell$ and $R(\Omega_i^-) = \{\mathcal{P}_i\}$, then $\Omega_i^+ = \Omega_{i+1}^-$ and $R(\Omega_{i+1}^-) = \{\mathcal{P}_{i+1}\}$. (i.e., $\mathcal{P}_{i+1}$ is the only releaser at Ω_{i+1}^-)

(d) If $i = \ell$, then $\Omega_i^+ = \Omega_{i+1}^-$ and $R(\Omega_{i+1}^-) = \{\mathcal{P}_{i+1}\}$.

(e) For all $t \in [\Omega_i^-, \Omega_i^+)$, $R(t) = \{\mathcal{P}_i\}$. (i.e., $\mathcal{P}_i$ is the only releaser throughout Ω_i)

Proof. **Proof of (a):** If the `PawnSet.promote()` operation at time γ returns value $\langle\perp, \perp\rangle$, then from Claims 4.4.23(g) and 4.4.23(j) it follows that no promotion occurs during T, which is a contradiction to $\ell \geq 1$. Thus, the `PawnSet.promote()` operation at time γ returns a non-$\langle\perp, \perp\rangle$ value. By definition γ is the point when the first

$\mathsf{PawnSet}.\texttt{promote()}$ operation occurs, and Ω_1^- is the point when the first promotion occurs and $\mathcal{P}_1$ is the process promoted at Ω_1^-. Then $\gamma = \Omega_1^-$, and $\mathcal{P}_1$ is the first promoted process. From Claim 4.4.23(e), $\mathcal{B}$ is the only releaser of L at the point in time immediately before time γ. Then from Claim 4.4.23(i) it follows that $\mathcal{B}$ promotes $\mathcal{P}_1$ at time $\gamma = \Omega_1^-$, and $\mathcal{B}$ ceases to be a releaser of L at γ, therefore $R(\gamma) = \{\mathcal{P}_1\}$. From Claim 4.4.23(f) it follows that $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ at Ω_1^-.

From an inspection of the code, a $\mathsf{PawnSet}.\texttt{reset()}$ is executed only in line 67, and it can be executed only after a $\mathsf{PawnSet}.\texttt{promote()}$ is executed in line 65. Since γ is the first point when a $\mathsf{PawnSet}.\texttt{promote()}$ is executed, it follows that no $\mathsf{PawnSet}.\texttt{reset()}$ operation was executed during $[I_0^-, \gamma]$.

Proof of (b): Since $R(\Omega_i^-) = \{\mathcal{P}_i\}$, and Ω_i^+ is the point when $\mathcal{P}_i$ ceases to be a releaser of L, for all $t \in [\Omega_i^-, \Omega_i^+)$, $\{\mathcal{P}_i\} \subseteq R(t)$. To show that for all $t \in [\Omega_i^-, \Omega_i^+)$, $R(t) = \{\mathcal{P}_i\}$, we need to show that no other process becomes a releaser of L, during $[\Omega_i^-, \Omega_i^+)$. Suppose some process $q = \mathcal{P}_i$ becomes a releaser of L some time during that interval. Since $\Omega_i^- > \Omega_1^- = \gamma > I_2^-$, from Claim 4.4.22(c) it follows that $\mathcal{P}_i$ promotes q during $[\Omega_i^-, \Omega_i^+)$. Then from Claim 4.4.16, $\mathcal{P}_i$'s cease-release event $\pi_{\mathcal{P}_i}$ occurs during $[\Omega_i^-, \Omega_i^+)$, and thus $\mathcal{P}_i$ ceases to be a releaser of L during $[\Omega_i^-, \Omega_i^+)$. Hence a contradiction.

Proof of (c): Since $i < \ell$, it follows that there exists a process $\mathcal{P}_{i+1}$ that becomes a releaser of L during T. By definition, $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ are the i-th and $(i+1)$-th promoted processes during T, respectively. Since $\Omega_{i+1}^- > \Omega_i^- > \Omega_1^- = \gamma > I_2^-$, from Claim 4.4.22(c) it follows that no other process becomes a releaser after $\mathcal{P}_i$ became a releaser and before $\mathcal{P}_{i+1}$ becomes a releaser, i.e., during $[\Omega_i^-, \Omega_{i+1}^-]$. Moreover, since $R(\Omega_i^-) = \{\mathcal{P}_i\}$, it follows that the next process to be promoted, i.e., $\mathcal{P}_{i+1}$, is promoted by the only releaser of L, $\mathcal{P}_i$. Then from Claim 4.4.16, it follows that $\mathcal{P}_i$ promotes $\mathcal{P}_{i+1}$ by executing a $\mathsf{PawnSet}.\texttt{promote()}$ in line 65 that returns $\langle \mathcal{P}_{i+1}, s \rangle$, where $s \in \mathbb{N}$, and event $\pi_{\mathcal{P}_i}$ occurs at $t_{\mathcal{P}_i}^{65}$. Then $\mathcal{P}_i$ ceases to be a releaser of L at $t_{\mathcal{P}_i}^{65}$ and thus $\Omega_i^+ = t_{\mathcal{P}_i}^{65}$. Since Ω_{i+1}^-

is the point when $\mathcal{P}_{i+1}$ becomes a releaser of L, it follows that $\Omega_i^+ = \Omega_{i+1}^-$, and thus $R(\Omega_{i+1}^-) = \{\mathcal{P}_{i+1}\}$.

Proof of (d): We prove by induction that for all $k < \ell$, $R(\Omega_{k+1}^-) = \{\mathcal{P}_{k+1}\}$ and $\Omega_k^+ = \Omega_{k+1}^-$.

Basis ($k = 1$) From Part (a), $\mathcal{P}_1$ is the only releaser of L at Ω_1^-, and clearly $\ell > k = 1$. Then from Part (c), $\Omega_1^+ = \Omega_2^-$ and $R(\Omega_2^-) = \{\mathcal{P}_2\}$.

Induction step ($k > 1$) By definition $\mathcal{P}_k$ is the promoted process at Ω_k^-, and since $|R(\Omega_{k-1}^+)| = 1$ and $\Omega_{k-1}^+ = \Omega_k^-$ (by the induction hypothesis), it follows that $\mathcal{P}_k$ is the only releaser of L at Ω_k^-. Then from Part (c), $\Omega_k^+ = \Omega_{k+1}^-$ and $R(\Omega_{k+1}^-) = \{\mathcal{P}_{k+1}\}$.

Proof of (e): From Part (a), $R(\Omega_1^-) = \{\mathcal{P}_1\}$, and thus from Part (b), for all $t \in [\Omega_1^-, \Omega_1^+)$, $R(t) = \{\mathcal{P}_1\}$. From Part (d), for all $i > 1$, $R(\Omega_i^-) = \{\mathcal{P}_i\}$, and thus from Part (b), for all $t \in [\Omega_i^-, \Omega_i^+)$, $R(t) = \{\mathcal{P}_i\}$. Hence, our claim follows. $\square$

Claim 4.4.26. *If $I_2 = \varnothing$ and $R(I_0^-) = \varnothing$ and at time I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \bot$ and $\mathsf{PawnSet}$ is candidate-empty, then the following claims hold for all $i \in \{1, \ldots, \ell\}$:*

(a) A $\mathsf{PawnSet}$`.reset()` operation is not executed during $[I_0^-, \Omega_i^-]$.

(b) $\mathcal{P}_i$ executes lines of code of $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$ starting with line 2 as depicted in Figure 4.15.

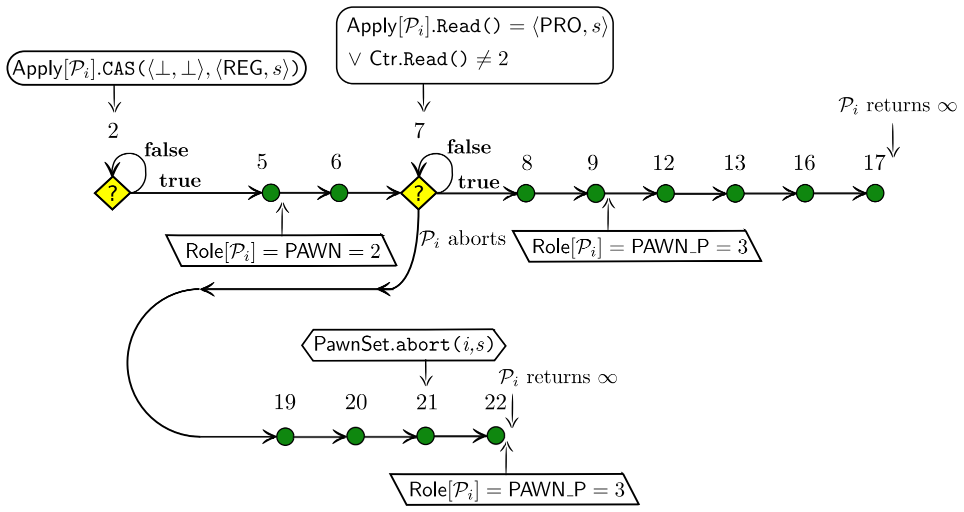
(c) $\mathcal{P}_i$'s call to $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$ returns ∞, and $\mathcal{P}_i$ finishes $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$ during T, and $\mathsf{Role}[\mathcal{P}_i] = \mathsf{PAWN\_P}$ when $\mathcal{P}_i$'s call to $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$ returns.

(d) Exactly one cease-release event among $\pi_{\mathcal{P}_i}$ and $\theta_{\mathcal{P}_i}$ occurs during $\mathcal{P}_i$'s call to $\texttt{doPromote}_{\mathcal{P}_i}\texttt{()}$.

(e) $\mathcal{P}_i$ executes lines of code of $\texttt{release}_{\mathcal{P}_i}\texttt{()}$ starting with line 34 as depicted in Figure 4.16.

(f) $\mathcal{P}_i$ does not write to $\mathsf{Sync1}$ or $\mathsf{Sync2}$ during $[\Omega_i^-, \Omega_i^+]$.

(g) $t_{\mathcal{P}_i}^2 < \Omega_i^- < t_{\mathcal{P}_i}^{34-} < \Omega_i^+ < t_{\mathcal{P}_i}^{49}$ and $\Omega_i^+ \leq I_2^+$.



Apply[$\mathcal{P}_i$].CAS(⟨⊥,⊥⟩,⟨REG,s⟩)
Apply[$\mathcal{P}_i$].Read() = ⟨PRO,s⟩ ∨ Ctr.Read() ≠ 2
$\mathcal{P}_i$ returns ∞
2
false
true
5
6
7
false
true
8
9
12
13
16
17
Role[$\mathcal{P}_i$] = PAWN = 2
$\mathcal{P}_i$ aborts
Role[$\mathcal{P}_i$] = PAWN_P = 3
PawnSet.abort(i,s)
$\mathcal{P}_i$ returns ∞
19
20
21
22
Role[$\mathcal{P}_i$] = PAWN_P = 3

Figure 4.15: $\mathcal{P}_i$'s call to $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$

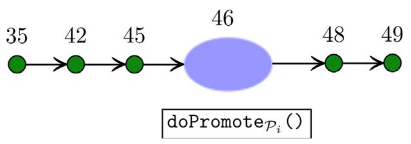


Figure 4.16: $\mathcal{P}_i$'s call to $\texttt{release}_{\mathcal{P}_i}\texttt{()}$

(h) If $i = \ell$, then a PawnSet.reset() *operation is not executed during* $[I_0^-, \Omega_i^+]$.

(i) Throughout $[\gamma, \Omega_\ell^+]$, Sync1 = Sync2 = ⊥.

(j) If $\ell > 1$, $I_2^+ = \Omega_\ell^+ = t_{\mathcal{P}_\ell}^{68}$.

(k) For all $t \in [\gamma, I_2^+)$, $|R(t)| = 1$.

(l) $R(I_2^+) = \varnothing$ *and at* I_2^+, Sync1 = Sync2 = ⊥ *and* PawnSet *is candidate-empty.*

Proof. **Proof of (a)-(h):** We prove Parts (a)-(h) by induction on i. First, we prove Part (a) for $i = 1$. Second, we show that if Part (a) is true for a fixed i, then Parts (b)-(h) are true for i. Finally, we show that if Parts (a)-(h) are true for i, then Part (a) is true for $i + 1$, thus completing the proof.

From Claim 4.4.25(a), no PawnSet.reset() operation has been executed during $[I_0^-, \Omega_1^-]$. Hence, Part (a) for $i = 1$ holds.

Now we show that if Part (a) is true for a fixed i, then Parts (b)-(h) follow for i.

Proof of Parts (b) and (c) if Part (a) for i is true: Let q be the process that promotes $\mathcal{P}_i$ at Ω_i^-. Then q's PawnSet.promote() operation in line 65 returned value $\langle \mathcal{P}_i, s\rangle$, where $s \in \mathbb{N}$, and $\Omega_i^- = t_q^{65}$. Then from the semantics of the PawnSet object it follows that the $\mathcal{P}_i$-th entry of PawnSet was $\langle \mathsf{REG}, s\rangle = \langle 1, s\rangle$ immediately before Ω_i^-. Then from Claim 4.4.3(b) it follows that some process (say r) executed a PawnSet.collect(A) operation in line 55 where $A[\mathcal{P}_i] = s$. Then from the code structure, r read $\mathsf{apply}[\mathcal{P}_i] = \langle \mathsf{REG}, s\rangle$ in line 52. By Claim 4.4.4(a) $\mathsf{apply}[\mathcal{P}_i]$ is set to value $\langle \mathsf{REG}, s\rangle$ only by process $\mathcal{P}_i$ when it executes a successful $\mathsf{apply}[\mathcal{P}_i]$.CAS($\langle \perp, \perp\rangle, \langle \mathsf{REG}, s\rangle$), therefore $\mathcal{P}_i$ executed the same and broke out of the spin loop of line 2. Note that $t_{\mathcal{P}_i}^2 < t_r^{52} < \Omega_i^- = t_q^{65}$.

Since $\mathsf{Ctr} = 0$ throughout I_0, $\mathsf{Ctr} = 1$ throughout I_1 and $\mathsf{Ctr} = 2$ throughout I_2, it follows that Ctr is increased only at points I_1^- and I_2^- during T. Since $\mathcal{K}$ and $\mathcal{Q}$ are the first two releasers of L and they increased Ctr to 1 and 2, respectively, at I_1^- and I_2^-, respectively, it follows that no other process apart from $\mathcal{K}$ and $\mathcal{Q}$ increases the value of Ctr during T. Since $\Omega_i^- \geq \Omega_1^- = \gamma > I_2^-$ (by Claims 4.4.23(a) and 4.4.23(d) and 4.4.25(a)), $\mathcal{P}_i$ becomes a releaser of L only after I_2^- (the point at which $\mathcal{Q}$ became a releaser of L). Thus, $\mathcal{P}_i$ is not among the first two releasers of L, thus $\mathcal{P}_i \notin \{\mathcal{K}, \mathcal{Q}\}$. Then it follows that $\mathcal{P}_i$ does not increase Ctr. Therefore $\mathcal{P}_i$'s Ctr.inc() operation in line 5 returns value $2 = \mathsf{PAWN}$, and thus $\mathcal{P}_i$ sets $\mathsf{Role}[\mathcal{P}_i]$ to $2 = \mathsf{PAWN}$ in line 5. Then from the code structure $\mathcal{P}_i$ satisfies the if-condition of line 6 and proceeds to spin in line 7.

Case a - $\mathcal{P}_i$ receives a signal to abort while busy-waiting in line 7: Then $\mathcal{P}_i$ stops spinning in line 7 and executes $\mathtt{abort}_{\mathcal{P}_i}$(). Since $\mathcal{P}_i$ last set $\mathsf{Role}[\mathcal{P}_i]$ to PAWN in line 5, it then follows from the code structure that $\mathcal{P}_i$ proceeds to execute lines 18-20, and satisfies the if-condition of line 20, and then executes a PawnSet.abort($\mathcal{P}_i, s$) operation in line 21.

Since a PawnSet.reset() operation has not been executed during $[I_0^-, \Omega_i^-]$, from

Claim 4.4.24, it follows that $\mathcal{P}_i$ did not execute a $\mathsf{PawnSet}.\mathtt{abort}(\mathcal{P}_i, s)$ operation in line 21 during $[I_0^-, \Omega_i^-]$, thus $t_{\mathcal{P}_i}^{21} > \Omega_i^-$. Since $\mathcal{P}_i$ has exclusive-registration access to $\mathsf{PawnSet}$ during $[\Omega_i^-, \Omega_i^+]$, and p has not executed any of its cease-release events or reset $\mathsf{PawnSet}$ during $[t_{\mathcal{P}_i}^2, t_{\mathcal{P}_i}^{21}]$, and $t_{\mathcal{P}_i}^2 < \Omega_i^-$, it then follows that $\mathsf{PawnSet}$ was not reset during $[\Omega_i^-, t_{\mathcal{P}_i}^{21}]$. Then since the $\mathcal{P}_i$-th entry of $\mathsf{PawnSet}$ was last changed to $\langle \mathsf{PRO}, s\rangle = \langle 2, s\rangle$ at Ω_i^-, it remains $\langle \mathsf{PRO}, s\rangle$ throughout $[\Omega_i^-, t_{\mathcal{P}_i}^{21}]$. Then $\mathcal{P}_i$'s $\mathsf{PawnSet}.\mathtt{abort}(\mathcal{P}_i, s)$ operation in line 21 returns `false` by the semantics of the $\mathsf{PawnSet}$ object. Then p satisfies the if-condition of line 21, proceeds to set $\mathsf{Role}[\mathcal{P}_i]$ to $\mathsf{PAWN\_P}$ in line 22, and then returns ∞ from its call to $\mathtt{abort}_{\mathcal{P}_i}()$ and $\mathtt{lock}_{\mathcal{P}_i}()$.

Case b - $\mathcal{P}_i$ does not receive a signal to abort while busy-waiting in line 7:

Recall that process q promotes $\mathcal{P}_i$ at Ω_i^- by executing a $\mathsf{PawnSet}.\mathtt{promote}()$ operation in line 65 that returns value $\langle \mathcal{P}_i, s\rangle$, where $s \in \mathbb{N}$. Since processes in the system continue to take steps, process q sets its local variable j to value $\mathcal{P}_i$ in line 65, and proceeds to fail the if-condition of line 66, and then executes line 70 where $\langle j, seq\rangle = \langle \mathcal{P}_i, s\rangle$. Then q executes a $\mathsf{apply}[\mathcal{P}_i].\mathtt{CAS}(\langle \mathsf{REG}, s\rangle, \langle \mathsf{PRO}, s\rangle)$ operation in line 70.

Recall that process r read value $\mathsf{apply}[\mathcal{P}_i] = \langle \mathsf{REG}, s\rangle$ in line 52 and $t_{\mathcal{P}_i}^2 < t_r^{52} < \Omega_i^- = t_q^{65}$. From an inspection of the code, $\mathsf{apply}[\mathcal{P}_i]$ can change from value $\langle \mathsf{REG}, s\rangle$ only to value $\langle \mathsf{PRO}, s\rangle$ and from value $\langle \mathsf{PRO}, s\rangle$ only to value $\langle \perp, \perp\rangle$. Also, $\mathsf{apply}[\mathcal{P}_i]$ can be changed from $\langle \mathsf{PRO}, s\rangle$ to $\langle \perp, \perp\rangle$, only if p executes line 32 or 49. Since p is spinning in line 7 it follows that a $\mathsf{apply}[\mathcal{P}_i].\mathtt{CAS}(\langle \mathsf{PRO}, s\rangle, \langle \perp, \perp\rangle)$ operation is not executed during (Ω_i^-, t_q^{70}), and thus $\mathsf{apply}[\mathcal{P}_i] = \langle \mathsf{REG}, s\rangle$ throughout (Ω_i^-, t_q^{70}). Therefore, q executes a successful $\mathsf{apply}[\mathcal{P}_i].\mathtt{CAS}(\langle \mathsf{REG}, s\rangle, \langle \mathsf{PRO}, s\rangle)$ operation in line 70, and thus $\mathsf{apply}[\mathcal{P}_i] = \langle \mathsf{PRO}, s\rangle$ at t_q^{70}.

Since $\mathcal{P}_i$ is busy-waiting in line 7 for $\mathsf{apply}[\mathcal{P}_i]$ to change to $\langle \mathsf{PRO}, s\rangle$, it then follows that $\mathcal{P}_i$ busy-waits throughout (Ω_i^-, t_q^{70}), and reads $\mathsf{apply}[\mathcal{P}_i] = \langle \mathsf{PRO}, s\rangle$ when it executes line 7 for the first time after t_q^{70}. Then $\mathcal{P}_i$ breaks out of the spin loop, and then from the code structure, $\mathcal{P}_i$ proceeds to set $\mathsf{Role}[\mathcal{P}_i]$ to $\mathsf{PAWN\_P}$ in line 9, breaks out of the role-loop in line 12, executes line 13 and fails the if-condition of line 13, and executes lines 16-17, and returns from $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$ in line 17 with value ∞. Note that $\Omega_i^- < t_{\mathcal{P}_i}^9$.

Proof of Parts (d), (e) and (f) if Part (a) for i is true: Since $\mathcal{P}_i$ is the only releaser of L throughout $[\Omega_i^-,\Omega_i^+)$ (Claim 4.4.25(e)), it follows from Claim 4.4.13 that $\mathcal{P}_i$ has exclusive write-access to objects $\mathsf{Sync1}$ and $\mathsf{Sync2}$ and exclusive registration-access to $\mathsf{PawnSet}$ throughout $[\Omega_i^-,\Omega_i^+)$.

Since $\mathcal{P}_i$ returns from its call to $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$ with value ∞ (by Part (c)), $\mathcal{P}_i$ executes a call to $\texttt{release}_{\mathcal{P}_i}\texttt{()}$ (follows from conditions b and c).

Since $\mathsf{Role}[\mathcal{P}_i] = \mathsf{PAWN\_P}$ when $\mathcal{P}_i$'s call to $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$ returns (by Part (c)), $\mathsf{Role}[\mathcal{P}_i] = \mathsf{PAWN\_P}$ at $t_{\mathcal{P}_i}^{34-}$. Since $\mathsf{Role}[\mathcal{P}_i]$ is unchanged during $[t_{\mathcal{P}_i}^{34}, t_{\mathcal{P}_i}^{49-}]$ (follows from Claim 4.4.2(b)), it follows from the code structure that during $\mathcal{P}_i$'s call to $\texttt{release}_{\mathcal{P}_i}\texttt{(}j\texttt{)}$, $\mathcal{P}_i$ only executes lines 34-35, 42 and 45-50. Then Figure 4.16 follows.

From an inspection of Figures 4.15 and 4.16, $\mathcal{P}_i$ does not execute a call to $\texttt{helpRelease}_{\mathcal{P}_i}\texttt{()}$ or execute a $\mathsf{Ctr}.\texttt{CAS(}1,0\texttt{)}$ operation in line 36 during $\texttt{release}_{\mathcal{P}_i}\texttt{()}$. Then from Claims 4.4.5(a) and 4.4.5(b) $\mathcal{P}_i$'s cease-release events $\phi_{\mathcal{P}_i}$ and $\tau_{\mathcal{P}_i}$ do not occur. Since $\mathcal{P}_i$ executes a call to $\texttt{doPromote}_{\mathcal{P}_i}\texttt{()}$ only in line 46, it follows from Claim 4.4.17 that exactly one cease-release event among $\pi_{\mathcal{P}_i}$ and $\theta_{\mathcal{P}_i}$ occurs during $\mathcal{P}_i$'s call to $\texttt{doPromote}_{\mathcal{P}_i}\texttt{()}$. Hence, Part (d) follows. Then Ω_i^+ is the point when cease-release event $\pi_{\mathcal{P}_i}$ or $\theta_{\mathcal{P}_i}$ occurs. From an inspection of Figures 4.15 and 4.16 and the code, it is clear that $\mathcal{P}_i$ does not change $\mathsf{Sync1}$ or $\mathsf{Sync2}$ during $\texttt{lock}_{\mathcal{P}_i}\texttt{()}$ and $\texttt{release}_{\mathcal{P}_i}\texttt{(.)}$ Therefore, $\mathcal{P}_i$ does not change $\mathsf{Sync1}$ or $\mathsf{Sync2}$ during $[\Omega_i^-, \Omega_i^+]$.

Proof of Part (g) if Part (a) for i is true: As argued in Part (b) and (c), $t_{\mathcal{P}_i}^5 < \Omega_i^-$, and $\Omega_i^- < t_{\mathcal{P}_i}^9$ or $\Omega_i^- < t_{\mathcal{P}_i}^{21}$. Since $t_{\mathcal{P}_i}^9 < t_{\mathcal{P}_i}^{34}$ and $t_{\mathcal{P}_i}^{21} < t_{\mathcal{P}_i}^{34}$, it then follows that $t_{\mathcal{P}_i}^5 < \Omega_i^- < t_{\mathcal{P}_i}^{34}$.

From Part (d), exactly one cease-release event among $\pi_{\mathcal{P}_i}$ and $\theta_{\mathcal{P}_i}$ occurs during $\mathcal{P}_i$'s call to doPromote$_{\mathcal{P}_i}$(). If cease-release event $\theta_{\mathcal{P}_i}$ occurs then Ω_i^+ is the point when $\mathcal{P}_i$'s cease-release event $\theta_{\mathcal{P}_i}$ occurs,i.e, $\Omega_i^+ = t^{68}_{\mathcal{P}_i}$. Then $\mathcal{P}_i$ changes Ctr to 0 and the Ctr-cycle interval T ends at $\Omega_i^+ = t^{68}_{\mathcal{P}_i} = I_2^+$.

If cease-release event $\pi_{\mathcal{P}_i}$ occurs then Ω_i^+ is the point when $\mathcal{P}_i$'s cease-release event $\pi_{\mathcal{P}_i}$ occurs,i.e, $\Omega_i^+ = t^{65}_{\mathcal{P}_i} < I_2^+$.

Since $\mathcal{P}_i$ calls doPromote$_{\mathcal{P}_i}$() only in line 46 (by inspection of Figure 4.16), it then follows that $\Omega_i^+ \in \;\; t^{65}_{\mathcal{P}_i}, t^{68}_{\mathcal{P}_i} \;\; < t^{49}_{\mathcal{P}_i}$. Thus, Part (g) holds.

Proof of Part (h) if Part (a) for i is true: As argued in Part (f), exactly one cease-release event among $\pi_{\mathcal{P}_i}$ and $\theta_{\mathcal{P}_i}$ occurs during $\mathcal{P}_i$'s call to doPromote$_{\mathcal{P}_i}$(). If cease-release event $\theta_{\mathcal{P}_i}$ occurs then Ω_i^+ is the point when $\mathcal{P}_i$'s cease-release event $\theta_{\mathcal{P}_i}$ occurs,i.e, $\Omega_i^+ = t^{68}_{\mathcal{P}_i}$. Then $\mathcal{P}_i$ changes Ctr to 0 and the Ctr-cycle interval T ends at $\Omega_i^+ = t^{68}_{\mathcal{P}_i}$, and thus $\ell = i$. This is a contradiction to the assumption $i = \ell$, hence $\mathcal{P}_i$'s cease-release event $\pi_{\mathcal{P}_i}$ occurs during $\mathcal{P}_i$'s call to doPromote$_{\mathcal{P}_i}$(). Then Ω_i^+ is the point when $\mathcal{P}_i$'s cease-release event $\pi_{\mathcal{P}_i}$ occurs,i.e, $\Omega_i^+ = t^{65}_{\mathcal{P}_i}$. From an inspection of Figures 4.15 and 4.16 and the code, it follows that $\mathcal{P}_i$ does not execute a PawnSet.reset() operation during $[t^{2}_{\mathcal{P}_i}, t^{46-}_{\mathcal{P}_i}]$, and $\mathcal{P}_i$ calls doPromote$_{\mathcal{P}_i}$() only in line 46. Since $\Omega_i^+ = t^{65}_{\mathcal{P}_i}$, from an inspection of the code of doPromote$_{\mathcal{P}_i}$(), $\mathcal{P}_i$ does not execute a PawnSet.reset() operation during $[t^{65-}_{\mathcal{P}_i}, t^{68-}_{\mathcal{P}_i}]$. Then $\mathcal{P}_i$ does not execute a PawnSet.reset() operation during $[\Omega_i^-, \Omega_i^+]$.

Since $\mathcal{P}_i$ is the only releaser of L throughout $[\Omega_i^-,\Omega_i^+)$ (Claim 4.4.25(e)), it follows from Claim 4.4.13 that $\mathcal{P}_i$ has exclusive registration-access to PawnSet throughout $[\Omega_i^-,\Omega_i^+)$. Then since no PawnSet.reset() operation was executed during $[I_0^-, \Omega_i^-]$, and $\mathcal{P}_i$ does not execute a PawnSet.reset() operation during $[\Omega_i^-, \Omega_i^+]$, it follows that no PawnSet.reset() operation is executed during $[I_0^-, \Omega_i^+]$. Hence, Part (h) holds.

Finally, we show that if Parts (a)-(h) are true for i, then Part (a) is true for $i + 1$, thus completing the proof. From Part (h) for i, no PawnSet.reset() operation has been executed during $[I_0^-, \Omega_i^+]$. From Claim 4.4.25(d), $\Omega_i^+ = \Omega_{i+1}^-$. Then Part (a) for $i + 1$

holds.

Proof of (i): From Claim 4.4.25(a), $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ at $\Omega_1^- = \gamma$. From Claims 4.4.25(a) and 4.4.25(d), it follows that $\gamma = \Omega_1^- < \Omega_1^+ = \Omega_2^- < \Omega_2^+ = \Omega_3^- \ldots < \Omega_{\ell-1}^+ = \Omega_\ell^- < \Omega_\ell^+$.

From Claim 4.4.25(e), for all $t \in [\Omega_i^-, \Omega_i^+)$, $R(t) \in \{\mathcal{P}_i\}$. Then $\mathcal{P}_i$ has exclusive write-access to Sync1 and Sync2 throughout $[\Omega_i^-, \Omega_i^+)$. Since $\mathcal{P}_i$ does not change Sync1 or Sync2 during $[\Omega_i^-, \Omega_i^+]$ (Part (f)), it then follows that $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ throughout $[\Omega_1^-, \Omega_\ell^+] = [\gamma, \Omega_\ell^+]$.

Proof of (j): As argued in Part (f), exactly one cease-release event among $\pi_{\mathcal{P}}$ and $\theta_{\mathcal{P}}$ occurs during $\mathcal{P}_\ell$'s call to $\mathtt{doPromote}_{\mathcal{P}}\ \mathtt{()}$. If cease-release event $\pi_{\mathcal{P}}$ occurs then $\mathcal{P}_\ell$ promotes some process, and thus the number of processes that get promoted during T is larger than ℓ, which contradicts the definition of ℓ. Hence, cease-release event $\theta_{\mathcal{P}}$ occurs during $\mathtt{doPromote}_{\mathcal{P}}\ \mathtt{()}$ and Ω_ℓ^+ is the point when cease-release event $\theta_{\mathcal{P}}$ occurs,i.e, $\Omega_\ell^+ = t_{\mathcal{P}}^{65}$. Since Ctr is changed from 2 to 0 when $\theta_{\mathcal{P}}$ occurs, the Ctr-cycle interval T ends at $\Omega_\ell^+ = t_{\mathcal{P}}^{68}$, and thus $I_2^+ = \Omega_\ell^+ = t_{\mathcal{P}}^{68}$.

Proof of (k) and (l):

Case a - $\ell = 0$: Consider the first PawnSet.promote() operation at γ. Since $\ell = 0$, the PawnSet.promote() operation at γ returns value $\langle \perp, \perp \rangle$. Then from Claim 4.4.23(j), it follows that $\mathcal{B}$'s cease-release event $\theta_{\mathcal{B}}$ occurs at $t' = t_{\mathcal{B}}^{68} \geq \gamma$, and throughout $[\gamma, t']$ no process is promoted, and for all $t \in [\gamma, t')$, $R(t) = \{\mathcal{B}\}$. Since Ctr is changed from 2 to 0 when $\theta_{\mathcal{B}}$ occurs, the Ctr-cycle interval T ends at $t' = t_{\mathcal{B}}^{68}$, and thus $I_2^+ = t_{\mathcal{B}}^{68} = t'$. Then for all $t \in [\gamma, t') = [\gamma, I_2^+)$, $|R(t)| = 1$.

From an inspection of Figure 4.14 and the code, it follows that $\mathcal{B}$ executed a PawnSet.reset() operation in line 67 during $[\gamma, t']$, and thus PawnSet is candidate-empty immediately after. Since for all $t \in [\gamma, t')$, $R(t) = \{\mathcal{B}\}$, $\mathcal{B}$ has exclusive registration-access to PawnSet throughout $[\gamma, t')$ (follows from Claim 4.4.13). Then it follows that PawnSet is candidate-empty at $t' = I_2^+$.

Since for all $t \in [\gamma, t')$, $R(t) = \{\mathcal{B}\}$, $\mathcal{B}$ has exclusive write-access to $\mathsf{Sync1}$ and $\mathsf{Sync2}$ throughout $[\gamma, t')$ (follows from Claim 4.4.13). Since $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ at γ (Claim 4.4.23(f)), and $\mathcal{B}$ does not write to $\mathsf{Sync1}$ and $\mathsf{Sync2}$ during $[\gamma, t']$, it follows that $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ throughout $[\gamma, t'] = [\gamma, I_2^+]$.

Case b - $\ell \geq 1$: From Part (j), $I_2^+ = \Omega_\ell^+ = t_\mathcal{P}^{68}$. Then from Part (i), it follows that $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ throughout $[\gamma, I_2^+]$, and from Claim 4.4.25(e), it follows that for all $t \in [\Omega_1^-, \Omega_\ell^+] = [\gamma, I_2^+)$, $|R(t)| = 1$. Since $\mathcal{P}_\ell$ ceases to be a releaser of L at Ω_ℓ^+, $R(I_2^+) = \varnothing$.

Since $\Omega_\ell^+ = t_\mathcal{P}^{68}$, $\mathcal{P}_i$ executed line 68 and before that line 67. Hence, $\mathcal{P}_\ell$ executed a $\mathsf{PawnSet.}\mathtt{reset()}$ operation at $t_\mathcal{P}^{67} < \Omega_\ell^+$. Since $t_\mathcal{P}^{67} > t_\mathcal{P}^{34-}$ and $t_\mathcal{P}^{34-} > \Omega_\ell^-$ (by Part (g)), it follows that $t_\mathcal{P}^{67} > \Omega_\ell^-$. Hence, $\mathcal{P}_\ell$ executed a $\mathsf{PawnSet.}\mathtt{reset()}$ operation at $t_\mathcal{P}^{67} \in [\Omega_\ell^-, \Omega_\ell^+]$. Since $\mathcal{P}_\ell$ is the only releaser of L throughout $[\Omega_\ell^-, \Omega_\ell^+)$ (Claim 4.4.25(e)), it follows from Claim 4.4.13 that $\mathcal{P}_\ell$ has exclusive registration-access to $\mathsf{PawnSet}$ throughout $[\Omega_\ell^-, \Omega_\ell^+)$. Then it follows that $\mathsf{PawnSet}$ is candidate-empty at $\Omega_\ell^- = I_2^+$. □

Claim 4.4.27. *$R(I_0^-) = \varnothing$ and at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty for any Ctr-cycle interval T during history H.*

Proof. Let T^k denote the k-th Ctr-cycle interval T during history H. We give a proof by induction over the integer k. **Basis -** At I_0^- for T^1, the claim holds trivially since all variables are at their initial values ($\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty).

Induction Step - By the induction hypothesis, at I_0^- for T^{k-1}, $R(I_0^-) = \varnothing$, and $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty. Since T^k begins immediately after T^{k-1} ends, to prove our claim we need to show that, when T^{k-1} ends, there are no releasers of L and $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty. The time interval T^{k-1} ends either at time I_1^+ or time I_2^+.

Case a - T^{k-1} ends at time I_1^+: Then $I_2 = \varnothing$. From Claim 4.4.21(f) it follows that $\mathcal{K}$ is the only releaser of L during I_1. Since $I_2 = \varnothing$, it then follows from Claim 4.4.21(e), that $\mathcal{K}$'s $\mathsf{Ctr}.\mathtt{CAS}(1, 0)$ operation in line 36 is successful, and the interval I_1 as well as T^{k-1} ends at time $t_{\mathcal{K}}^{36}$. Then $\mathcal{K}$'s cease-release event $\phi_{\mathcal{K}}$ occurs at $t_{\mathcal{K}}^{36} = I_1^+$, and thus there are no releasers of L immediately after T^{k-1} ends. And from Claim 4.4.21(g), it follows that $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty when T^{k-1} ends.

Case b - T^{k-1} ends at time I_2^+: Then $I_2 = \varnothing$. Then our proof obligation follows immediately from Claim 4.4.26(l). □

Note that in the following claims, notations I_0, I_1, I_2, λ, γ, Ω_i, $\mathcal{K}$, $\mathcal{Q}$ and $\mathcal{P}_i$ are defined relative to a Ctr-cycle interval, as was defined previously in pages 106, 115 and 121. The exact Ctr-cycle interval is clear from the context of the discussion.

Lemma 4.4.2. *The mutual exclusion property holds during history H.*

Proof. For the purpose of a contradiction assume that at time t, two processes (say p and q) are poised to execute a call to $\mathsf{L}.\mathtt{release()}$. From Claim 4.4.11(b), it follows that both p and q are releasers of L at t. Consider the Ctr-cycle interval T such that $t \in T$.

From Claim 4.4.27 it follows that at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty, and $R(I_0^-) = \varnothing$. Then from Claims 4.4.21(a), 4.4.21(f), 4.4.23(a), 4.4.23(e) and 4.4.23(k), it follows that during T, lock L has two releasers only during $[I_2^-, \lambda)$. Then $t \in [I_2^-, \lambda)$. Also from Claim 4.4.23(a), for all $t \in [I_2^-, \lambda)$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$. Then $\{p, q\} = \{\mathcal{K}, \mathcal{Q}\}$. Let $p = \mathcal{K}$ and $q = \mathcal{Q}$ without loss of generality.

Recall that I_2^- is the point in time when $\mathcal{Q}$ increases Ctr from 1 to 2 and sets $\mathsf{Role}[\mathcal{Q}]$ to QUEEN in line 5. Since q's call to $\mathtt{lock()}$ returned a non-$\perp$ value, it follows from an inspection of Figure 4.10, that $\mathcal{Q}$ returned either in line 17 or line 27. Then $\mathcal{Q}$ either read a non-$\perp$ value from $\mathsf{Sync1}$ in line 14 or $\mathcal{Q}$ failed the $\mathsf{Sync1}.\mathtt{CAS}(\perp, \infty)$ operation in line 26. Since $\mathsf{Sync1} = \perp$ at I_2^- (by Claim 4.4.22(a)), and $I_2^- = t_{\mathcal{Q}}^5$, it then follows that $\mathsf{Sync1}$ is changed to a non-$\bot$ value during $[I_2^-, t]$. Clearly, $\mathcal{Q}$ does not change $\mathsf{Sync1}$ during $[I_2^-, t]$.

Recall that I_1^- is the point in time when $\mathcal{K}$ increases Ctr from 0 to 1 and sets $\mathsf{Role}[\mathcal{K}]$ to KING in line 5. It follows from an inspection of Figure 4.8, that $\mathcal{K}$ does not change $\mathsf{Sync1}$ during $\mathtt{lock}_{\mathcal{K}}\mathtt{()}$, and thus during $[I_1^-, t]$. Since $\mathsf{Sync1}$ is changed to a non-$\bot$ only by a releaser of L (by Claim 4.4.13) and $\mathsf{Sync1} = \bot$ at I_2^-, and the only releasers of L during $[I_2^-, t]$ do not change $\mathsf{Sync1}$, it then follows that $\mathsf{Sync1} = \bot$ throughout $[I_2^-, t]$. Hence, a contradiction. $\square$

Claim 4.4.28. *Consider an arbitrary* Ctr*-cycle interval* T*.*

(a) If p *is collected during* T *and* p *does not abort, then* p *is promoted and notified during* T*.*

(b) If $\mathsf{apply}[p] = \langle \mathsf{REG}, s\rangle$ *at* I_0^-*, where* $s \in N$*, and* p *does not abort and* p *does not increase* Ctr*, then* p *is notified during* T*.*

Proof. **Proof of (a):** From Claim 4.4.27 it follows that at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \bot$ and $\mathsf{PawnSet}$ is candidate-empty, and $R(I_0^-) = \varnothing$. Then from Claim 4.4.23(k), it follows that exactly one call to `doCollect()` is executed during T by a process $q \in \{\mathcal{K}, \mathcal{Q}\}$. Since processes are collected only during a call to `doCollect()`, $q \in \{\mathcal{K}, \mathcal{Q}\}$ collects p during $\mathtt{doCollect}_q\mathtt{()}$ during T. And q does so by executing a $\mathsf{PawnSet}.\mathtt{collect}(A)$ operation in line 55, where $A[p] = s \in \mathbb{N}$, and sets the p-th entry of $\mathsf{PawnSet}$ to $\langle \mathsf{REG}, s\rangle$. Since a $\mathsf{PawnSet}.\mathtt{promote()}$ that returns $\langle \bot, \bot\rangle$ is executed at $t_{\mathcal{P}}^{65}$ during T, it then follows from the semantics of the $\mathsf{PawnSet}$ object that p was promoted during T. Then $p = \mathcal{P}_i$, for some $i \le \ell$. Note that T does not end during $[\Omega_i^-, \Omega_i^+)$.

We now show that p is also notified of its promotion during T. The process (say r) that promoted p by executing a $\mathsf{PawnSet}.\mathtt{promote()}$ operation in line 65, also goes on to notify p of its promotion by executing a $\mathsf{apply}[p].\mathtt{CAS}(\langle \mathsf{REG}, s\rangle, \langle \mathsf{PRO}, s\rangle)$ operation in line 70. Since p does not abort, it follows from an inspection of Figure 4.15 and the code, that p spins on $\mathsf{apply}[p]$ in line 7 until its notification. Then p executes line 9 at $t_p^9 > t_r^{70} > t_r^{65} = \Omega_i^-$. Since $t_p^9 < \Omega_i^+$ and T does not end before Ω_i^+, it follows that p is notified during T.

Proof of (b): Since p does not increase Ctr it follows that p reads $\mathsf{Ctr} = 2$ every time it executes a $\mathsf{Ctr}.\texttt{inc()}$ operation in line 5, and sets $\mathsf{Role}[p] = \mathsf{PAWN}$ in line 5. Then p satisfies the if-condition of line 6 and spins on variables $\mathsf{apply}[p]$ and Ctr in line 7. Since Ctr is only changed to 0 at the end of T, it follows that $\mathsf{Ctr} = 2$ throughout $[t_p^5, I_2^+)$. Then p busy-waits in the spin loop of line 7 until the end of T, or if it reads value $\langle \mathsf{PRO}, s\rangle$, for some $s \in \mathbb{N}$, from $\mathsf{apply}[p]$ in line 7 during T. Now, $\mathsf{apply}[p]$ is changed to value $\langle \mathsf{PRO}, s\rangle$ by some process other than p, only if that process notifies p, i.e., executes a successful $\mathsf{apply}[p].\texttt{CAS(}\langle \mathsf{REG}, s\rangle\texttt{,}\langle \mathsf{PRO}, s\rangle\texttt{)}$ operation in line 70. We now show that p is notified during T.

From Claim 4.4.27 it follows that at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty, and $R(I_0^-) = \varnothing$. Then from Claim 4.4.23(k), it follows that exactly one call to $\texttt{doCollect()}$ is executed during T by a process $q \in \{\mathcal{K}, \mathcal{Q}\}$. Consider the point when q reads $\mathsf{apply}[p]$ in line 52. If q reads a value different from $\langle \mathsf{REG}, s\rangle$, then some process must have notified p during $[t_p^2, t_q^{52}]$, and since $I_0^- < t_p^2$ and $t_q^{52} \in T$, our claim holds. If q reads the value $\langle \mathsf{REG}, s\rangle$ from $\mathsf{apply}[\mathsf{p}]$, then q collects p during T by executing a $\mathsf{PawnSet}.\texttt{collect(}A\texttt{)}$ operation, where $A[p] = s$, in line 55 during T. Thus, our claim follows from Part (a). □

Claim 4.4.29. *If p registered itself in line 2, and incurred $\mathcal{O}(1)$ RMRs in the process, and p does not abort, and all processes in the system continue to take steps, then*

(a) p finishes its call to $\texttt{lock}_p\texttt{()}$ and returns a non-$\perp$ value.

(b) p incurs $\mathcal{O}(1)$ RMRs in expectation during its call to $\texttt{lock}_p\texttt{()}$.

Proof. **Proof of (a) and (b):** From an inspection of the code of $\texttt{lock}_p\texttt{()}$, p incurs a constant number of RMRs while executing all other lines of $\texttt{lock}_p\texttt{()}$ except while busy-waiting in lines 2, 7 and 14.

Consider p's call to $\texttt{lock}_p\texttt{()}$. By assumption of the claim, p registered itself in line 2 by executing a successful $\mathsf{apply}[p].\texttt{CAS(}\langle\bot,\bot\rangle,\langle\mathsf{REG},s\rangle\texttt{)}$ operation in line 2, and incurred $\mathcal{O}(1)$ RMRs in the process. Then p proceeds to execute a $\mathsf{Ctr}.\texttt{inc()}$ operation in line 5, and stores the returned value in $\mathsf{Role}[p]$. A $\mathsf{Ctr}.\texttt{inc()}$ operation returns values in $\{\mathsf{KING},\mathsf{QUEEN},\mathsf{PAWN},\bot\}$. If it returns $\bot$, p repeats the role-loop, and executes another $\mathsf{Ctr}.\texttt{inc()}$ operation in line 5. From Claim 4.1.2, it follows that p repeats the role-loop only a constant number of times before its $\mathsf{Ctr}.\texttt{inc()}$ operation returns a non-$\bot$ value.

Case a - p executes a $\mathsf{Ctr}.\texttt{inc()}$ operation in line 5 that returns KING. Then p sets $\mathsf{Role}[p] = \mathsf{KING}$ in line 5. Then from the code structure p does not busy-wait on any variables, and proceeds to return ∞ in line 17, and thus incurs only $\mathcal{O}(1)$ RMRs. Hence, (a) and (b) hold.

Case b - p executes a $\mathsf{Ctr}.\texttt{inc()}$ operation in line 5 that returns QUEEN. Then p increments Ctr from 1 to 2 in line 5 and sets $\mathsf{Role}[p] = \mathsf{QUEEN}$ in line 5. Then from the code structure p proceeds to busy-wait on $\mathsf{Sync1}$ in line 14. Since p increased Ctr from 1 to 2, $t_p^{14} = I_2^-$ for some Ctr-cycle interval T. From Claim 4.4.27 it follows that at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \bot$ and $\mathsf{PawnSet}$ is candidate-empty, and $R(I_0^-) = \varnothing$. Then from Claim 4.4.22(g) it follows that p does not starve in line 14. Since p does not abort, it follows from an inspection of Figure 4.10 and the code, that p returns a non-$\bot$ value in line 17, and p does not change $\mathsf{Sync1}$. Hence, we have shown that Part (a) holds. Apart from p, the only releasers of L during T are $\{\mathcal{K},\mathcal{P}_1,\ldots,\mathcal{P}_\ell\}$, where ℓ is the number of promotions during T. From an inspection of Figures 4.8, 4.9, 4.15, 4.16 and the code, it follows that only $\mathcal{K}$ possibly writes a non-$\bot$ value to $\mathsf{Sync1}$ during T in line 37. Since $\mathsf{Sync1}$ is written to only be a releaser of L, and $t_p^{14} \in T$, it then follows that $\mathsf{Sync1}$ is changed to a non-$\bot$ value at most once during T. Then p incurs at most one RMR while busy-waiting on Sync1. Hence, we have shown that Part (b) holds.

Case c - p executes a `Ctr.inc()` operation in line 5 that returns PAWN. Then p found Ctr to be 2 in line 5 and set Role$[p]$ = PAWN in line 5. Then from the code structure p proceeds to busy-wait on apply$[p]$ and Ctr in line 7.

We now show that p does not starve while busy-waiting in line 7. Since Ctr $= 2$ at t_p^5, it follows that $t_p^5 \in T$ for some Ctr-cycle interval T.

Subcase (i) - apply$[p] = \langle$REG$, s\rangle$ at I_0^- during T, for some $s \in \mathbb{N}$. Then from Claim 4.4.28(b), p is notified during T. Since p is notified during T and p does not abort, it follows that p does not change apply$[p]$, and thus apply$[p]$ is changed from $\langle$REG$, s\rangle$ to $\langle$PRO$, s\rangle$ when p is notified. Since apply$[p]$ is changed from $\langle$PRO$, s\rangle$ to some other value only by p, it then follows that apply$[p]$ remains $\langle$PRO$, s\rangle$ when p reads apply$[p]$ for the first time after p was notified. Then p incurs one RMR when it reads apply$[p]$ in line 7 after its notification, breaks out of the spin loop of line 7, proceeds to satisfy the if-condition of line 8, and sets Role$[p]$ = PAWN_P in line 9, and proceeds to return ∞ in line 17. Then we have shown Parts (a) and (b) hold.

Subcase (ii) - apply$[p] = \langle$REG$, s\rangle$ at I_0^- during T, for some $s \in \mathbb{N}$. Consider the only call to `doCollect()` during T by $q \in \{\mathcal{K}, \mathcal{Q}\}$. If p registered itself (i.e., executed its apply$[p]$.`CAS(`$\langle\bot, \bot\rangle$,$\langle$REG$, s\rangle$`)` operation in line 2) before q reads apply$[p]$ in line 52 during `doCollect`$_q$`()`), then q collects p during T. Then from Claim 4.4.28(a), p is collected and promoted during T, and eventually notified. Then Parts (a) and (b) hold as argued in **Subcase (i)**.

If p registers itself after q attempts to acknowledge p during T, then no process changes apply$[p]$ during T. Then p continues to busy-wait in line 7, until the Ctr-cycle interval T ends and Ctr is reset to 0.

If Ctr is increased to 2 before p reads Ctr again in line 7, then let T' be the Ctr-cycle interval that starts when Ctr was reset to 0 at the end of T. Since $\mathsf{apply}[p]$ was changed to a non-$\langle\mathsf{REG}, s\rangle$ value before the start of T', it follows that $\mathsf{apply}[p] = \langle\mathsf{REG}, s\rangle$ at the start of T'. Then from Claim 4.4.28(b), p is acknowledged, collected, promoted during T', and eventually notified. Then Parts (a) and (b) hold as argued in **Subcase (i)**.

If $\mathsf{Ctr} = 2$ when p reads Ctr again in line 7, then p incurs one RMR in line 7, breaks out of the spin loop, and proceeds to execute line 8. If p satisfies the if-condition of line 8, then p has been acknowledged during some Ctr-cycle interval T''. Then from Claim 4.4.28(a), p is collected, promoted during T'', and eventually notified. Then Parts (a) and (b) hold as argued in **Subcase (i)**. If p fails the if-condition of line 8, then p proceeds to repeat the role-loop. Consider p's second iteration of the role-loop. If p sets $\mathsf{Role}[p] = \{\mathsf{KING}, \mathsf{QUEEN}\}$ in line 5, then Parts (a) and (b) hold as argued in **Case a** and **Case b**. If p sets $\mathsf{Role}[p] = \mathsf{PAWN}$ in line 5, then it follows that $t_p^5 \in T'''$, for some Ctr-cycle interval T''', such that $\mathsf{apply}[p] = \langle\mathsf{REG}, s\rangle$ at I_0^- for T'''. Parts (a) and (b) hold as argued in **Case c(i)**. □

Lemma 4.4.3. *If all processes in the system continue to take steps and p does not abort, then*

(a) p finishes its call to `lock`$_p$`()` *and returns a non-$\perp$ value.*

(b) p incurs $\mathcal{O}(1)$ RMRs in expectation during its call to `lock`$_p$`()`*.*

Proof. From an inspection of `lock`$_p$`()`, p incurs a constant number of RMRs while executing all other lines of `lock`$_p$`()` except while busy-waiting in lines 2, 7 and 14.

Consider p's call to `lock`$_p$`()`. Process p first attempts to register itself in line 2, by attempting to execute an $\mathsf{apply}[p]$.`CAS(`$\langle\perp,\perp\rangle$,$\langle\mathsf{REG}, s\rangle$`)` operation. Now, $\mathsf{apply}[p]$ is changed from $\langle\perp,\perp\rangle$ to a non-$\langle\perp,\perp\rangle$ value only by p (Claim 4.4.4(a)). If $\mathsf{apply}[p] = \langle\perp,\perp\rangle$ at t_p^{2-}, then p executes a successful $\mathsf{apply}[p]$.`CAS(`$\langle\perp,\perp\rangle$, $\langle\mathsf{REG}, s\rangle$`)` operation in line 2 and incurs only one RMR. Then our claims follow immediately from Claims 4.4.29(a) and 4.4.29(b).

If $\mathsf{apply}[p] = \langle\bot, \bot\rangle$ at t_p^{2-}, it follows that some process p' executed a successful $\mathsf{apply}[p]$.`CAS(`$\langle\bot, \bot\rangle, \langle\mathsf{REG}, s'\rangle$`)` in line 2 during $\texttt{lock}_p$`()`, and $\mathsf{apply}[p] = \langle\bot, \bot\rangle$ throughout $[t_{p'}^2, t_p^{2-}]$. Since calls to $\texttt{lock}_p$`()` are not executed concurrently, it follows that p' has completed its call to $\texttt{lock}_p$`()` during $[t_{p'}^2, t_p^{2-}]$.

Case 1 - p''s call to $\texttt{lock}_p$`()` returned $\bot$. Then it follows from the code structure that p' executed a call to $\texttt{abort}_p$`()` and returned from line 18 or 33. Since p executed a successful $\mathsf{apply}[p]$.`CAS(`$\langle\bot, \bot\rangle, \langle\mathsf{REG}, s'\rangle$`)` in line 2, p' could not have aborted while busy-waiting on line 2, and thus p' aborted while busy-waiting in line 7 or 14. Then p' executed line 3, and set its local variable $p'.flag$ to **true**, and thus p could not have returned $\bot$ from line 18 during $\texttt{abort}_p$`()`. Then p' returned $\bot$ in line 33, and thus p' executed operations $\mathsf{apply}[p]$.`CAS(`$\langle\mathsf{REG}, s'\rangle$,$\langle\mathsf{PRO}, s'\rangle$`)` (in line 19), and $\mathsf{apply}[p]$.`CAS(`$\langle\mathsf{PRO}, s'\rangle$,$\langle\bot, \bot\rangle$`)` (in line 32). Since, $\mathsf{apply}[p]$ can be changed from $\langle\mathsf{REG}, s'\rangle$ only to $\langle\mathsf{PRO}, s'\rangle$, and from $\langle\mathsf{PRO}, s'\rangle$ only to $\langle\bot, \bot\rangle$, it then follows that p' executes a successful $\mathsf{apply}[p]$.`CAS(`$\langle\mathsf{PRO}, s'\rangle$,$\langle\bot, \bot\rangle$`)` (in line 32). Then p' eventually resets $\mathsf{apply}[p]$ during its $\texttt{lock}_p$`()` call. Since $\mathsf{apply}[p] = \langle\bot, \bot\rangle$ throughout $[t_{p'}^2, t_p^{2-}]$ and p' completed its call to $\texttt{lock}_p$`()` during $[t_{p'}^2, t_p^{2-}]$, we have a contradiction.

Case 2 - p''s call to $\texttt{lock}_p$`()` returned a non-$\bot$ value. Then from the code structure p' executed operations $\mathsf{apply}[p]$.`CAS(`$\langle\mathsf{REG}, s'\rangle$,$\langle\mathsf{PRO}, s'\rangle$`)` (in line 16 or line 19) before returning from its call to $\texttt{lock}_p$`()`. Since $\mathsf{apply}[p]$ can be changed from $\langle\mathsf{REG}, s'\rangle$ only to $\langle\mathsf{PRO}, s'\rangle$, and from $\langle\mathsf{PRO}, s'\rangle$ only to $\langle\bot, \bot\rangle$ and only by a process with pseudo-ID p, it then follows that $\mathsf{apply}[p] = \langle\mathsf{PRO}, s'\rangle$ when p''s $\texttt{lock}_p$`()` returns. Then it also follows that $\mathsf{apply}[p] = \langle\mathsf{PRO}, s'\rangle$ until a process with pseudo-ID p executes an $\mathsf{apply}[p]$.`CAS(`$\langle\mathsf{PRO}, s'\rangle$,$\langle\bot, \bot\rangle$`)` operation.

Since p' won the lock L, it follows that some process, say r, eventually executes a call to $\texttt{release}_p(j)$, for some integer j. Since a call to $\texttt{release}_p(j)$ is wait-free and all processes continue to take steps, it follows that eventually r executes lines 48 and 49 where it reads value $\langle \mathsf{PRO}, s' \rangle$ from $\mathsf{apply}[p]$ in line 48 and resets $\mathsf{apply}[p]$ with a $\mathsf{apply}[p].\texttt{CAS}(\langle \mathsf{PRO}, s' \rangle, \langle \bot, \bot \rangle)$ operation in line 49. Since p does not abort, and no other process calls $\texttt{lock}_p\texttt{()}$ concurrently, it then follows that eventually p executes a successful $\mathsf{apply}[p].\texttt{CAS}(\langle \bot, \bot \rangle, \langle \mathsf{REG}, s \rangle)$ operation in line 2. Since $\mathsf{apply}[p]$ changed only once from $\langle \mathsf{PRO}, s' \rangle$ to $\langle \bot, \bot \rangle$ while p busy-waited in line 2, it follows that p incurs $\mathcal{O}(1)$ RMRs during the entire process. Then our claims follow immediately from Claims 4.4.29(a) and 4.4.29(b). □

Lemma 4.4.4. *The abort-way is wait- free.*

Proof. The abort-way is defined to be all steps taken by a process (say p) after it receives a signal to abort and breaks out of one of the busy-wait cycles of lines 2, 7 or 14. After p breaks out of one of the busy-wait cycles of lines 2, 7 or 14 p executes a call to $\texttt{abort}_p\texttt{()}$. If p's call to $\texttt{abort}_p\texttt{()}$ returns $\bot$, then p's passage ends, or else p's $\texttt{lock}_p\texttt{()}$ returns non-$\bot$ value and p calls $\texttt{release}_p\texttt{()}$ and p's passage ends when the $\texttt{release}_p\texttt{()}$ method returns. Since $\texttt{abort}_p\texttt{()}$ and $\texttt{release}_p\texttt{()}$ are both wait-free (by Lemma 4.4.1), our claim follows. □

Lemma 4.4.5. *The starvation freedom property holds during history H.*

Proof. Consider a process p that begins to execute its passage. From Lemma 4.4.3(a), it follows that if p does not abort during $\texttt{lock}_p\texttt{()}$ and all processes continue to take steps then p eventually returns from $\texttt{lock}_p\texttt{()}$ with a non-$\bot$ value. Then p eventually calls $\texttt{release}_p\texttt{()}$, and since $\texttt{release}_p\texttt{()}$ is wait-free, p eventually completes its passage. If p receives a signal to abort during $\texttt{lock}_p\texttt{()}$, then p executes its abort-way. Since the abort-way is wait-free (by Lemma 4.4.4), p eventually completes its passage. □

Lemma 4.4.6. *If a call to* $\texttt{release}_p\texttt{(}j\texttt{)}$ *returns* **true**, *then there exists a concurrent call to* `lock()` *that eventually returns* j.

Proof. The only operations that write a value to $\mathsf{Sync1}$ are $\mathsf{Sync1}.\texttt{CAS(}\perp,\infty\texttt{)}$ in line 26, and $\mathsf{Sync1}.\texttt{CAS(}\perp,j\texttt{)}$ in line 37. From Claim 4.4.13, $\mathsf{Sync1}$ is written to only by a releaser of L. From Claim 4.4.27 it follows that at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty, and $R(I_0^-) = \varnothing$. Then from Claims 4.4.21(a), 4.4.21(f), 4.4.23(a), 4.4.23(e), 4.4.26(k), and 4.4.26(l), the only releasers of L during a Ctr-cycle interval T, are $\{\mathcal{K}, \mathcal{Q}, \mathcal{P}_1, \ldots, \mathcal{P}_\ell\}$. Then from an inspection of Figures 4.8, 4.8, 4.10, 4.11, 4.15 and 4.16, it follows that only $\mathcal{K}$ and $\mathcal{Q}$ can write to $\mathsf{Sync1}$ during Ctr-cycle interval T.

Since p returns **true**, it then follows from an inspection of the code that p executed a successful $\mathsf{Sync1}.\texttt{CAS(}\perp,j\texttt{)}$ operation in line 37, and thus failed the $\mathsf{Ctr}.\texttt{CAS(}1,0\texttt{)}$ operation in line 36 and $\mathsf{Role}[p] = \mathsf{KING}$ at t_p^{36}. Then $p = \mathcal{K}$ for some Ctr-cycle interval T. Since $\mathcal{K}$ failed the $\mathsf{Ctr}.\texttt{CAS(}1,0\texttt{)}$ operation in line 36, it then follows that Ctr was increased to 1 by process $\mathcal{Q}$ during T, and $I_2^- = t_{\mathcal{Q}}^5 < t_{\mathcal{K}}^{36}$. Since $I_1^- = t_{\mathcal{K}}^5$ and $I_1^- < I_2^-$, it then follows that $\mathcal{Q}$'s $\texttt{lock}_{\mathcal{Q}}\texttt{()}$ call is concurrent to $\mathcal{K}$'s $\texttt{release}_{\mathcal{K}}\texttt{(}j\texttt{)}$ call.

From Claim 4.4.27 it follows that at I_0^-, $\mathsf{Sync1} = \mathsf{Sync2} = \perp$ and $\mathsf{PawnSet}$ is candidate-empty, and $R(I_0^-) = \varnothing$. Then from Claim 4.4.23(a), $\mathsf{Sync1} = \perp$ at I_2^-, and $\mathcal{K}$ and $\mathcal{Q}$ are the only two releasers of L during $[I_2^-, \lambda)$, where λ is the first point in time when T is changed to a non-$\perp$ value, and $\lambda = \texttt{min(}t_{\mathcal{K}}^{56}\texttt{,}t_{\mathcal{Q}}^{56}\texttt{)}$.

Now, $\mathsf{Sync1}$ is reset only in line 58, and since $t_{\mathcal{K}}^{58} > t_{\mathcal{K}}^{56} \geq \lambda$ and $t_{\mathcal{Q}}^{58} > t_{\mathcal{Q}}^{56} \geq \lambda$, it then follows that $\mathcal{K}$ and $\mathcal{Q}$ do not reset $\mathsf{Sync1}$ during $[I_2^-, \lambda]$. Since $\mathcal{K}$ and $\mathcal{Q}$ are the only processes with write-access to $\mathsf{Sync1}$, $\mathsf{Sync1}$ is not reset during $[I_2^-, \lambda]$.

Consider $\mathcal{Q}$'s `lock()` call (see Figure 4.10). Since $\mathcal{K}$ executed a successful $\mathsf{Sync1}.\texttt{CAS(}\perp,j\texttt{)}$ operation and $\mathsf{Sync1}$ is not reset during $[I_2^-, \lambda]$, it then follows that if $\mathcal{Q}$ executes the $\mathsf{Sync1}.\texttt{CAS(}\perp,\infty\texttt{)}$ operation in line 26, then the operation fails. From an inspection of Figure 4.10, $\mathcal{Q}$ either returns from its `lock()` call in line 17 or line 27. In both these lines, $\mathcal{Q}$ returns the non-$\perp$ value stored in Sync1. Since $\mathcal{K}$ is the only process apart from $\mathcal{Q}$ that can write to Sync1 $\mathcal{Q}$ returns the value j that $\mathcal{K}$ wrote during its $\texttt{release}_{\mathcal{K}}(j)$ call. □

The following theorem follows from Lemmas 4.4.1-4.4.6.

Theorem 4.4.1. *Object* RandALockArray *is an implementation of type* TransferableAbortableLock *and satisfies mutual exclusion, starvation freedom, bounded exit, and bounded abort properties against the weak adversary. The object requires $\mathcal{O}(n)$* `CAS` *objects and read-write registers.*

Now consider an implementation of object RandALockArray, where instance PawnSet is implemented using object $\mathsf{SFMSUnivConst}\langle\mathsf{AbortableProArray}_n\rangle$, and the operations in lines 55, 61, 65, 64, and 67 are executed using the `performFast()` method, while the operation in line 21 is executed using the `performSlow()`.

Claim 4.4.30. *Lines 64,65, 67 of* `doPromote()`*, all lines of* `doCollect()`*, and lines 57-62 are not executed concurrently.*

Proof. From Claim 4.4.11(b), it follows that only a releaser of L can execute any of these lines. From Claim 4.4.27 it follows that at I_0^-, Sync1 $=$ Sync2 $= \perp$ and PawnSet is candidate-empty, and $R(I_0^-) = \varnothing$. Then from Claims 4.4.21(a), 4.4.21(f), 4.4.23(a), 4.4.23(e), 4.4.26(k), and 4.4.26(l) it follows that L has more than one releaser only during $[I_2^-, \lambda)$ for some Ctr-cycle interval T. More specifically, there are two releasers of L only during $[I_2^-, \lambda)$, and the releasers are $\mathcal{K}$ and $\mathcal{Q}$. From Claim 4.4.23(k) it follows that a `doCollect()` is executed only by $\mathcal{K}$ or $\mathcal{Q}$ but not both. Then it follows immediately that lines of `doCollect()` are not executed concurrently. Since $\lambda = \texttt{min}(t_{\mathcal{K}}^{56},\, t_{\mathcal{Q}}^{56})$, it follows from an inspection of Figures 4.8, 4.9, 4.10, 4.11 and the code, that processes $\mathcal{K}$ and $\mathcal{Q}$ have not executed a call to `doPromote()` or lines 57-62 of `helpRelease()`, before $t^{56}_{\mathcal{K}}$ and $t^{56}_{\mathcal{Q}}$ respectively. Then none of the lines chosen in the claim are executed concurrently, and thus our claim holds. □

Lemma 4.4.7. *(a)* $\texttt{helpRelease}_p\texttt{()}$ *and* $\texttt{doPromote}_p\texttt{()}$ *have an RMR complexity of* $\mathcal{O}(1)$*.*

(b) $\texttt{doCollect}_p\texttt{()}$ *has an RMR complexity of* $\mathcal{O}(n)$*.*

(c) $\texttt{abort}_p\texttt{()}$ *has an RMR complexity of* $\mathcal{O}(n)$*.*

(d) If a call to $\texttt{release}_p\texttt{(}j\texttt{)}$ *returns* **true***, then* p *incurs* $\mathcal{O}(n)$ *RMRs during* $\texttt{release}_p\texttt{(}j\texttt{)}$*.*

(e) If a call to $\texttt{release}_p\texttt{(}j\texttt{)}$ *returns* **false***, then* p *incurs* $\mathcal{O}(1)$ *RMRs during* $\texttt{release}_p\texttt{(}j\texttt{)}$*.*

Proof. **Proof of (a) and (b):** As per the properties of object SFMSUnivConst$\langle$AbortableProArray$_n\rangle$ (Theorem 4.2.2), an operation performed using the `performFast()` method has $\mathcal{O}(1)$ RMR complexity, as long as it is not executed concurrently with another `performFast()` method call. Since PawnSet is an instance of object SFMSUnivConst$\langle$AbortableProArray$_n\rangle$, where operations in lines 55, 61, 65, 64, and 67 are executed using the `performFast()` method, and each of these operations are not executed concurrently (by Claim (4.4.30)), it then follows that all of these operations have $\mathcal{O}(1)$ RMR complexity. Then Part (a) follows immediately from an inspection of methods `helpRelease()` and `doPromote()`. Since method `doCollect()` has a loop of size n that incurs a constant number of RMRs in each iteration, Part (b) follows.

Proof of (c), (d) and (e): As per the properties of object SFMSUnivConst$\langle$AbortableProArray$_n\rangle$ (Theorem 4.2.2), an operation performed using the `performSlow()` method has $\mathcal{O}(n)$ RMR complexity, where n is the maximum number of processes that can access the object concurrently. Since the operation in line 21 is executed using the `performSlow()` method, the operation has $\mathcal{O}(n)$ RMR complexity. Since `helpRelease()` and `doPromote()` have an RMR complexity of $\mathcal{O}(1)$ (by Part (a)), and `doCollect()` has an RMR complexity of $\mathcal{O}(n)$ (by Part (b)), it then follows from an inspection of `abort()`, that a call to `abort()` has an RMR complexity of $\mathcal{O}(n)$. Thus Part (b) follows.

If a call to $\texttt{release}_p(j)$ returns **true**, then p does execute a call to $\texttt{doCollect}_p\texttt{()}$ in line 38, else it does not. Then from an inspection of $\texttt{release}_p(j)$, Parts (d) and (e) follow immediately. □

The following theorem follows from Theorem 4.4.1 and Lemma 4.4.7.

Theorem 4.4.2. *Object* RandALockArray*, where instance* PawnSet *is implemented using object* $\mathsf{SFMSUnivConst}\langle\mathsf{AbortableProArray}_n\rangle$*, is an implementation of type* TransferableAbortableLock *and satisfies the following properties against the weak adversary for the CC model:*

(a) Mutual exclusion, starvation freedom, bounded exit, and bounded abort.

(b) The abort-way has $\mathcal{O}(n)$ RMR complexity.

(c) If a process does not abort during a `lock()` *call, then it incurs $\mathcal{O}(1)$ RMRs in expectation during the call, otherwise it incurs $\mathcal{O}(n)$ RMRs in expectation during the call.*

(d) If a process' call to `release(`j`)` *returns* **false***, then it incurs $\mathcal{O}(1)$ RMRs during the call, otherwise it incurs $\mathcal{O}(n)$ RMRs during the call.*

The object requires $\mathcal{O}(n)$ `CAS` *objects and read-write registers.*

4.5 Tree based Randomized Abortable Lock

In this section we specify, implement, and prove properties of our N process abortable lock object RandALockTree, where N is the maximum number of processes that can access the lock concurrently. Object RandALockTree is a tree based randomized implementation of type AbortableLock. An algorithm that accesses an instance of object RandALockTree must satisfy the following:

Condition 4.5.1. (a) A process calls method `release()` if and only if its last access of the lock object was a successful `lock()` call.

(b) Methods $\texttt{lock}_p\texttt{()}$ and $\texttt{release}_p\texttt{()}$ are called only by process p, where $p \in \{0, \ldots, N-1\}$.

Since we have N processes, for convenience we assume (w.l.o.g.) that $N = \Delta^{\Delta-1}$ for some positive integer Δ. Then it follows that $\Delta = \Theta(\log N/ \log\log N)$. An execution of an algorithm that accesses an instance of an object RandALockTree where Condition 4.5.1 is satisfied, has the following properties:

(a) Mutual exclusion, starvation freedom, bounded abort and bounded exit hold.

(b) The abort-way has $\mathcal{O}(\Delta)$ RMR complexity.

(c) Process p incurs $\mathcal{O}(\Delta)$ RMRs in expectation during $\texttt{lock}_p\texttt{()}$.

(d) Process p incurs $\mathcal{O}(\Delta)$ RMRs during $\texttt{release}_p\texttt{()}$.

4.5.1 High Level Description

We now describe the implementation of object RandALockTree (see Figure 4.17). Similarly to many mutual exclusion locks, our object uses an arbitration tree. Each process starts at a leaf of the arbitration tree and moves up the tree to the root, locking nodes on its path. Once a process locks the root, it can enter the Critical Section. Each node of the arbitration tree contains a single instance of the abortable randomized lock object RandALockArray. An instance of object RandALockArray at every node serves as a lock object of the node, while also providing processes the ability to abort their attempt at any point in time during their ascent to the root node.

Shared Data Structure - The Arbitration Tree. The data structure underlying the algorithm is a complete Δ-ary tree $\mathcal{T}$ of height Δ with N leaves, called the arbitration tree. The internal nodes of the arbitration tree $\mathcal{T}$ is a structure of type Node that consists of a single instance of a RandALockArray object, L (see Figure 4.17). The N processes in the system line up as N unique leaf nodes, such that each process p is associated with a unique leaf leaf_p in the tree. We say that a node is at level i if its height is i, where the root has height Δ and the leaves of the tree have height 0.

Let path_p denote the path from leaf_p up to root, and h_u denote the height of node u. We assume that the tree structure $\mathcal{T}$ provides a function `nodeOnPath()`, such that, for a leaf node leaf and integer ℓ, the function `nodeOnPath`(leaf, ℓ) returns a pair $\langle u, i\rangle$, where u is the ℓ-th node on the path from leaf to the root node, and i is the index of the child node of u that lies on the path.

Owning and Transferring a node. Consider a node u on path_p of some process p. Process p is said to *capture* lock u.L if its call to u.L.`lock()` returns a non-$\perp$ value. Process p is said to *release* lock u.L if p executes a call to u.L.`release()`. Process p is said to *hand over* all nodes from node v to u on path_p to a process q, if p executes a v.L.`release(`j`)` call that returns **true**, where $j = \mathsf{h}_u \geq \mathsf{h}_v$ and q executes a concurrent v.L.`lock()` call that returns j. Process p hands over node u so that some other process can call u.L.`release()` on p's behalf. Recall that method `release()` of object RandALockArray has $\mathcal{O}(1)$ RMR complexity if the method returns **false**, and $\mathcal{O}(\Delta)$ RMR complexity if

```
Class RandALockTree
    define Node: struct {
        L: RandALockArray
    }
    shared:
        $\mathcal{T}$: complete $\Delta$-ary tree of height $\Delta$ and node type Node
    local:
        $v$: Node init $\perp$,
        $i, \ell, k$: int init $0$,
        $abort\_signal$: boolean init false,

Method lock$_p$()
    while $\ell < \mathcal{T}.height$ do
        $(v, i) \leftarrow \mathcal{T}$.nodeOnPath(leaf$_p$, $\ell + 1$)
        $val \leftarrow v$.L.lock$_i$()
        if $val = \infty$ then $\ell \leftarrow \ell + 1$
        if $val \notin \{\perp, \infty\}$ then $\ell \leftarrow val$
        if $abort\_signal$ = true then
            release$_p$()
            return $\perp$
        end
    end
    return $\infty$

Method release$_p$()
    while $k \leq \ell$ do
        $(v, i) \leftarrow \mathcal{T}$.nodeOnPath(leaf$_p$, $k$)
        if $v$.L.release$_i$($\ell$) then break
        $k \leftarrow k + 1$
    end
```

Figure 4.17: Implementation of Object RandALockTree

the method returns **true**. Handing over nodes helps bound the RMR complexity of the `release()` method of object $\mathsf{RandALockTree}$ to $\mathcal{O}(\Delta)$ RMR complexity. Process p starts to *own* node u when p captures $u.\mathsf{L}$ or when p is handed over node u from the previous owner of node u. Suppose p owns all nodes on path_p up to node u. Process p *ceases* to own node u if p releases $u.\mathsf{L}$ or if p hands over node u to some other process. We prove that at any point in time there is at most one process that owns a node u, and thus we refer to that process by owner_u.

Lock capture protocol - $\texttt{lock}_p\texttt{()}$. The lock capture protocol executed during $\texttt{lock}_p\texttt{()}$ is as follows. Process p climbs up the tree on its path starting from leaf node leaf_p to the root node of $\mathcal{T}$, and attempts to *capture* every node that it does not own, as long as p has not received a signal to abort. Process p attempts to capture a node u by executing a call to $u.\mathsf{L}.\texttt{lock()}$. If p's $u.\mathsf{L}.\texttt{lock()}$ call returns ∞ then p is said to have captured u, and if the call returns an integer j, then p is said to have been handed over all nodes from u to v on path_p, where $\mathsf{h}_v = j$. We ensure that $j \geq \mathsf{h}_u$.

Process p can enter its Critical Section when it owns the root node of $\mathcal{T}$. Process p may receive a signal to abort during a call to $u.\mathsf{L}.\texttt{lock()}$ as a result of which p's call to $u.\mathsf{L}.\texttt{lock()}$ returns either $\perp$ or a non-$\perp$ value. In either case, p then calls $\texttt{release}_p\texttt{()}$ to release all locks of nodes that p has captured in its passage, and then returns from its $\texttt{lock}_p\texttt{()}$ call with value $\perp$.

Lock release protocol - $\texttt{release}_p\texttt{()}$. An exiting process p *releases* all nodes that it owns during $\texttt{release}_p\texttt{()}$. Process p is said to *release* node u if p releases $u.\mathsf{L}$ (by executing $u.\mathsf{L}.\texttt{release()}$ call), or if p hands over node u to some other process. Recall that p hands over node u if p executes a $v.\mathsf{L}.\texttt{release(}j\texttt{)}$ call that returns **true** where $\mathsf{h}_v \leq \mathsf{h}_u \leq j$.

Let s be the height of the highest node p owns. During $\texttt{release}_p\texttt{()}$, p climbs up

$\mathcal{T}$ and calls u.L.`release`$_p$`(`s`)` at every node u that it owns, until a call returns **true**. If a u.L.`release`$_p$`(`s`)` call returns **false**, then p is said to have released lock u.L (and therefore released node u), and thus p continues on its path. If a u.L.`release`$_p$`(`s`)` call returns **true**, then p has handed over all remaining nodes that it owns to some process that is executing a concurrent u.L.`lock()` call at node u, and thus p does not release any more node.

Notice that our strategy to release node locks is to climb up the tree until all node locks are released or a hand over of remaining locks is made. Climbing up the tree is necessary (as opposed to climbing down) in order to hand over node locks to a process, say q, such that the handed over nodes lie on path$_q$. There is however a side effect of this strategy which is as follows: Suppose p owns nodes v and u on path$_p$ such that $\langle u, i\rangle =$ `nodeOnPath(`leaf$_p$,h$_u$`)` and v is the i-th child on node u. Now suppose p releases lock v.L at node v. Since the lock at node v is now released, some process $r = p$ may now capture lock v.L and then proceed to call u.L.`lock`$_i$`()`. If process p has not yet released u.L by completing its call to u.L.`release`$_i$`()`, then we have a situation where a call to u.L.`lock`$_i$`()` is made before a call to u.L.`release`$_i$`()` is completed. Since there can be at most one owner of lock v.L there can be at most one such call to u.L.`lock`$_i$`()` concurrent to u.L.`release`$_i$`()`. This is precisely the reason why we designed object RandALockArray to be accessed by at most $n + 1$ processes concurrently.

4.5.2 Implementation / Low Level Description

We now describe the implementation of object RandALockTree (see Figure 4.17).

Description of the `lock`$_p$`()` method. Suppose process p executes a call to `lock`$_p$`()`. With every iteration of the while-loop, process p captures at least one node on its path from leaf$_p$ to $\mathcal{T}$.root. Suppose p executes an iteration of while-loop (lines 1-10) and $\ell_p = k$ at line 1 for some arbitrary integer k. In line 2, process p determines the k-th node (say

u) on path_p and the index (say r) of u's child node that lies on path_p, and stores them in local variables v_p and i_p. The variables v_p and i_p are unchanged during the rest of the iteration. In line 3, process p attempts to capture $u.\mathsf{L}$, and thus node u by executing a call to $u.\mathsf{L}.\mathtt{lock()}$ with pseudo-ID r. If p's $u.\mathsf{L}.\mathtt{lock}_r\mathtt{()}$ returns an integer value (say j) then p has been transferred all nodes on its path up to height j (we ensure $j \geq \mathsf{h}_u$). If p's $u.\mathsf{L}.\mathtt{lock()}$ returns ∞ then p has captured lock $u.\mathsf{L}$. In lines 4 and 5, p stores the height of the highest captured node in its local variable ℓ_p. In line 6, p checks whether it has received a signal to abort. In this case p releases all its captured nodes by executing a call to $\mathtt{release}_p\mathtt{()}$ in line 7 and then returns from its call to $\mathtt{lock}_p\mathtt{()}$ in line 8 with value $\perp$. Otherwise p continues its while-loop. On completing its while-loop, p owns the root node, and thus returns with value ∞ in line 11 to indicate a successful $\mathtt{lock()}$ call.

Description of the $\mathtt{release}_p\mathtt{()}$ method. Suppose process p executes a call to $\mathtt{release}_p\mathtt{()}$. Let s be the highest node p owns at the beginning of $\mathtt{release}_p\mathtt{()}$. We later prove that $\mathsf{h}_s = \ell_p$. During an iteration of the while-loop (lines 12-16), process p either releases a node on its path from leaf_p to s, or p hands over all remaining nodes that it owns to some process.

Consider the execution of an iteration of the while-loop where $k_p = t$ at line 12 for some integer $t \leq \mathsf{h}_s$. In line 13, process p determines the t-th node (say u) on path_p and the index (say r) of u's child node that lies on path_p, and stores them in local variables v_p and i_p. In line 14, process p releases $u.\mathsf{L}$, and thus node u, by executing a call to $u.\mathsf{L}.\mathtt{release(}\mathsf{h}_s\mathtt{)}$ with pseudo-ID r. If p's $u.\mathsf{L}.\mathtt{release}_r\mathtt{(}\mathsf{h}_s\mathtt{)}$ returns **false** then p has successfully released lock $u.\mathsf{L}$, and thus node u. If p's $u.\mathsf{L}.\mathtt{release}_r\mathtt{(}\mathsf{h}_s\mathtt{)}$ returns **true** then p has successfully handed over all nodes from u to s on path_p to some process that is executing a concurrent call to $u.\mathsf{L}.\mathtt{lock()}$. If p has handed over all its nodes, then p breaks out of the while-loop in line 14, and returns from its call to $\mathtt{release}_p\mathtt{()}$. If p has not handed over all its nodes then p increases k_p in line 15 and continues its while-loop.

4.5.3 Analysis and Proofs of Correctness

In this section, we formally prove all properties of $\mathsf{RandALockTree}$ as stated in Subsection 4.5, for the CC model. We start by defining some notation and terminology.

Notations and Definitions. Let H be an arbitrary history of an algorithm that accesses an instance L of object $\mathsf{RandALockTree}$ where Condition 4.5.1 is satisfied. Consider an arbitrary node u on the tree $\mathcal{T}$. Let h_u denote the height of node u.

A node u is said to be *handed over* from process p to process q, when p executes a $v.\mathsf{L}.\mathtt{release}(j)$ call that returns **true**, where $j \geq \mathsf{h}_u > \mathsf{h}_v$ and q executes a concurrent $v.\mathsf{L}.\mathtt{lock()}$ call that returns j. Process p is said to start to *own* node u when p captures $u.\mathsf{L}$ or when it is handed over node u from the previous owner of node u. Process p *ceases* to own node u when p releases $u.\mathsf{L}$, or when p hands over node u to some other process.

Claim 4.5.1. *Consider an arbitrary process p and some node u on path_p.*

(a) If p executes a $u.\mathsf{L}.\mathtt{lock()}$ operation that returns value $j \notin \{\perp, \infty\}$, then $j \geq \mathsf{h}_u$.

(b) The value of ℓ_p is increased every time p writes to it.

(c) If $\ell_p = k$, then process p owns all nodes on path_p up to height k.

Proof. **Proof of (a) :** Then from the the properties of object $\mathsf{RandALockArray}$ (Theorem 4.4.2), it follows that some process (say q) executed a concurrent $u.\mathsf{L}.\mathtt{release}(j)$ operation. Then from the code structure, q executed a $u.\mathsf{L}.\mathtt{release}(j)$ in line 14, where $\ell_q = j$. Then q also executed a $\mathcal{T}.\mathtt{nodeOnPath}(\mathsf{leaf}_q, k)$ operation in line 13 that returned $\langle u, i\rangle$, for some i, such that $\mathsf{h}_u = k_q$ (from the semantics of the $\mathtt{nodeOnPath()}$ method). Since $j = \ell_q \geq k_q = \mathsf{h}_u$, our claim follows.

Proof of (b): Process p writes to its local variable ℓ_p only in lines 4 and 5. Clearly, p increases ℓ_p every time it executes line 4. Now, suppose p executes line 5 where it writes the value of val_p to ℓ_p, where $v_p = u$, for some node u. Since p satisfies the if-condition of line 5 and the RandALockArray method `lock()` only returns a value in $\{\bot, \infty\} \cup \mathbb{N}$, it follows that p's call to u.L.`lock()` returned a non-$\{\bot, \infty\}$ value. Then from Part (a), $val_p \geq \mathsf{h}_u$. Since p also executed a $\mathcal{T}$.`nodeOnPath`(leaf_p, b) operation in line 2, where $b = \ell_p + 1$ that returned $\langle u, i \rangle$, for some i, such that $\mathsf{h}_u = b$ (from the semantics of the `nodeOnPath()` method), it follows that $val_p \geq \mathsf{h}_u = \ell_p + 1$. Then, p increases ℓ_p when p writes val_p to ℓ_p in line 5.

Proof of (c): Let t^i be the point in time such that p writes to its local variable ℓ_p for the i-th time. We prove our claim by induction over i

Basis ($i = 0$): Since the initial value of ℓ_p is 0 and ℓ_p is written to for the first time only at $t^1 > t^0$, the claim holds.

Induction step ($i > 0$): Let the value of ℓ_p be j after the $(i-1)$-th write to it. Then from the induction hypothesis, p owns all nodes on path_p up to height j. Consider the iteration of the while-loop during which p writes to ℓ_p for the i-th time, and specifically the $\mathcal{T}$.`nodeOnPath`($\mathsf{leaf}_p, \ell + 1$) operation in line 2. Since $\ell_p = j$, at the beginning of this while-loop iteration, it follows from the semantics of the `nodeOnPath()` operation, that the operation returned the pair $\langle u, i \rangle$, for some i, where $\mathsf{h}_u = j + 1$. Now, process p writes to its local variable ℓ_p only in lines 4 and 5.

Case a - p writes to ℓ_p in line 4. Then p increased ℓ_p from j to $j+1$ in line 4. Then, to prove our claim we need to show that p owns the node with height $j+1$ on path_p. Since p satisfies the if-condition of line 4, it follows from the code structure that p's u.L.`lock()` method in line 3 returned the special value ∞, where $v_p = u$. Since $\mathsf{h}_u = j + 1$, and p successfully captured lock u.L, it follows that p owns the $j + 1$-th node on path_p.

Case b - p writes to ℓ_p in line 5. Let $val_p = x$ when p writes to ℓ_p in line 5. From Part (b), it follows that ℓ_p is increased every time it is written to, and therefore $val_p = x > \ell_p$ when p writes to ℓ_p in line 5. Thus, to prove our claim we need to show that p owns all nodes on path_p with heights in the range $\{j, \ldots, x\}$. Since p satisfies the if-condition of line 5 and the $\mathsf{RandALockArray}$ method `lock()` only returns a value in $\{\perp, \infty\} \cup \mathbb{N}$, it follows that p's call to $u.\mathsf{L}.\mathtt{lock()}$ returned a non-$\{\perp, \infty\}$ value. Thus, p has captured $u.\mathsf{L}$ and now owns node u. It also follows that p has been handed over all nodes on path_p with heights in the range $\{\mathsf{h}_u + 1, \ldots, x\}$. Since $\mathsf{h}_u = j$, our claim follows. □

A process is said to *attempt to capture node* u if it executes a $u.\mathsf{L}.\mathtt{lock()}$ method in line 3.

Claim 4.5.2. *(a) If two distinct processes p and q attempt to capture node v, then their local variables i have different values.*

(b) A node has at most one owner at any point in time.

Proof. We prove our claims for all nodes of height at most h, by induction over integer h.

Basis ($h = 1$) Consider an arbitrary node u of height 1, such that two distinct processes p and q attempt to capture node u. Then processes p and q executed a $\mathtt{nodeOnPath}(\langle \mathsf{leaf}_p, 1\rangle)$ and $\mathtt{nodeOnPath}(\langle \mathsf{leaf}_q, 1\rangle)$ in line 2, and received pairs $\langle u, i\rangle$ and $\langle u, j\rangle$, and set their local variables i_p and i_q to i and j respectively. Since p and q are distinct, leaf_p and leaf_q are distinct leaf nodes of tree T, and thus from the semantics of the `nodeOnPath()` method it follows that $i = j$, and thus Part (a) follows.

Consider an arbitrary node u of height 1. From Part (a), it follows that no two processes execute a concurrent call to $u.\mathsf{L}.\mathtt{lock}_i\mathtt{()}$ for the same i, and thus it follows from the mutual exclusion property of object $\mathsf{RandALockArray}$, that at most one process captures $u.\mathsf{L}$. By definition, a process can become an owner of node u only if it captures $u.\mathsf{L}$ or if it is handed over node u from some other process q. If a node u is handed over from some other process q, then q also ceases to be the owner of node u at that point, and thus the number of owners of u does not increase upon a hand over. Thus it follows that node u has at most one owner at any point in time, and thus Part (b) follows.

Induction Step ($h > 1$) Consider an arbitrary node u of height h, such that two distinct processes p and q attempt to capture node u. Then processes p and q executed a `nodeOnPath(`$\langle \mathsf{leaf}_p, h\rangle$`)` and `nodeOnPath(`$\langle \mathsf{leaf}_q, h\rangle$`)` in line 2, and received pairs $\langle u, i\rangle$ and $\langle u, j\rangle$, and set their local variables i_p and i_q to i and j, respectively. For the purpose of a contradiction, assume $i = j$. From the semantics of `nodeOnPath()` method, $i = j$ only if the $(h-1)$-th nodes on path_p and path_q are the same (say w). From the induction hypothesis of Part (b) for $h - 1$, w has at most one owner at any point in time. Since $\ell_p = \ell_q = h - 1$ when p and q attempt to capture node u, it follows from Claim 4.5.1(c), that p and q own all nodes up to height $h - 1$ on their individual paths path_p and path_q. Then p and q are both the owners of w – a contradiction. Thus, Part (a) follows.

Since Part (a) holds for h, Part (b) holds for h, as argued in the **Basis** case. □

Lemma 4.5.1. *The mutual exclusion property is satisfied during history H.*

Proof. Assume two processes p and q are in their Critical Section at the same time, i.e., both processes returned a non-$\perp$ value from their last `lock()` call. Then both processes executed line 11 and thus $\ell_p = \ell_q = \mathcal{T}.height$ holds. Then from Claim 4.5.1(c) it follows that both p and q own node $\mathcal{T}.\mathsf{root}$. But from Claim 4.5.2(b), at most one process may own $\mathcal{T}.\mathsf{root}$ at any point in time – a contradiction. □

Claim 4.5.3. *Process p repeats the while-loop in* `lock()` *at most Δ times.*

Proof. Consider an arbitrary process p that calls `lock()`. From the code structure of `lock()`, it follows that if p repeats an iteration of the while-loop then p either executed line 4 or line 5 in its previous iteration. Then it follows from Claim 4.5.1(b) that p increases ℓ_p every time it repeats an iteration of the while-loop. Since the height of the $\mathcal{T}$ is Δ, our claim follows. □

Lemma 4.5.2. *No process starves in history H.*

Proof. Since no two processes execute a concurrent call to u.L.`lock`$_i$`()` for the same i (from Claim 4.5.2 (a)), it follows from the starvation-freedom property of object RandALockArray, that a process does not starve during a call to u.L.`lock()` for some node u on its path.

Consider an arbitrary process p that calls `lock()`. Since p repeats the while-loop in `lock()` at most Δ times before returning from line 11 (follows from Claim 4.5.3), it follows that p starves only if p starves during a call to u.L.`lock()` in line 3 for some node u. As already argued, this cannot happen, and thus our claim follows. □

Lemma 4.5.3. *Process p incurs $\mathcal{O}(\Delta)$ RMRs during* `release`$_p$`()`.

Proof. Consider p's call to `release()`. Since $\ell_p \leq \mathcal{T}.height = \Delta$, it follows from an inspection of the code that during `release()`, p executes at most Δ calls to L.`release()` (in line 14), and at most one of the L.`release()` calls returns **true**. As per the properties of object RandALockArray (Theorem 4.4.2), a process incurs $\mathcal{O}(\Delta)$ RMRs during a call to L.`release()`, if the call returns **true**, otherwise $\mathcal{O}(1)$ RMRs. Then our claim follows immediately. □

Lemma 4.5.4. *Process p incurs $\mathcal{O}(\Delta)$ RMRs in expectation during* `lock`$_p$`()`.

Proof. A process may or may not receive a signal to abort during `lock`$_p$`()`.

Case a - p does not receive a signal to abort during `lock`$_p$`()`. As per the properties of object RandALockArray (Theorem 4.4.2), if a process does not receive a signal to abort during a call to L.`lock()`, then the process incurs $\mathcal{O}(1)$ RMRs in expectation during the call. Since p repeats the while-loop in `lock()` at most Δ times (by Claim 4.5.3), and p does not receive a signal to abort during `lock`$_p$`()`, it follows that p incurs $\mathcal{O}(\Delta)$ RMRs in expectation during `lock`$_p$`()`.

Case b - p receives a signal to abort during `lock`$_p$`()`. As per the properties of object RandALockArray (Theorem 4.4.2), if a process aborts during a call to `L.lock()`, then the process incurs $\mathcal{O}(\Delta)$ RMRs in expectation during the call. Since p repeats the while-loop in `lock()` at most Δ times (by Claim 4.5.3), and p executes at most one call to u.`L.lock()` after having received an abort signal, it follows that p incurs $\mathcal{O}(\Delta)$ RMRs in expectation during `lock`$_p$`()`. □

Lemma 4.5.5. *Method* `release()` *is wait-free.*

Proof. As per the bounded exit of object RandALockArray, method `release()` of the object is wait-free. Then our claim follows immediately from an inspection of the code of `release()`. □

Lemma 4.5.6. *The abort-way is wait-free and has $\mathcal{O}(\Delta)$ RMR complexity.*

Proof. The abort-way of a process p consists of the steps executed by the process after receiving a signal to abort and before completing its passage. From Lemma 4.5.5 and 4.5.3, method `release`$_p$`()` is wait-free, and has $\mathcal{O}(\Delta)$ RMR complexity. From Claim 4.5.3, a process repeats the while-loop in `lock`$_p$`()` at most Δ times. Then from an inspection of the code it follows that a process executes all steps during its passage in a wait-free manner, except the call to u.`L.lock()` in line 3, and that a process incurs at most $\mathcal{O}(\Delta)$ RMRs during all these steps.

To complete our proof we now show that if a process has received a signal to abort and it executes a call to u.`L.lock()` in line 3, for some node u, then the process executes u.`L.lock()` in a wait-free manner and incurs $\mathcal{O}(\Delta)$ RMR during the call, and does not call v.`L.lock()` for any other node v.

Suppose that p has received a signal to abort, and p executes a call to u.`L.lock()` call in line 3. Since p has received a signal to abort, it follows that p executes the abort-way of the node lock u.L. As per the properties of object RandALockArray (Theorem 4.4.2), its abort-way is wait-free and has $\mathcal{O}(\Delta)$ RMR complexity. Then p executes the u.L.`lock()` call in line 3 in a wait-free manner and incurs $\mathcal{O}(\Delta)$ RMR complexity. It then goes on to satisfy the if-condition of line 6, and executes a call to `release()` in line 7 and returns $\perp$ in line 8, thereby completing its abort-way. Thus, our claim holds. □

The following theorem follows from Lemmas 4.5.1-4.5.6.

Theorem 4.5.1. *Object* RandALockTree *is a starvation-free randomized abortable N process mutual exclusion lock for the CC model, where the RMR complexity of a passage is $\mathcal{O}(\log N/\log\log N)$, in expectation against the weak adversary. The algorithm requires $\mathcal{O}(N)$* `CAS` *objects and read-write registers.*

Chapter 5

Conclusion

We presented a randomized abortable mutual exclusion algorithm with $\mathcal{O}(\log N/\log\log N)$ expected RMRs, against a weak adversary for the CC model. Although we achieved the goal of our thesis, there is scope for future work. First, object $\mathsf{RandALockArray}$ uses sequence numbers to solve the classic ABA problem, and therefore the registers of the apply array are unbounded. Our universal construction object $\mathsf{SFMSUnivConst}\langle\rangle$ also uses unbounded registers for the same reason. A fix to the above problem could be to modify our algorithms to use `LL/SC` objects instead, which are inherently free from the ABA problem. Secondly, our abortable lock works against the weak adversary but not the adaptive adversary. Since object $\mathsf{RCAScounter}_2$ is the only place where coin flips (random choices) are made, a replacement of this component with one that works against an adaptive adversary and is equally efficient would fix this problem, although we are not sure if such a component can be designed. Thirdly, our lock is not adaptive. We feel that our lock $\mathsf{RandALockTree}$ can be modified to be adaptive by using techniques similar to the one presented in [38], where lock HWLock is modified to be adaptive, because our lock uses the same arbitration tree structure as that of HWLock. Fourthly, in this thesis we presented proofs that our algorithm works for the CC model, and we believe that the same algorithm can be modified to work for the DSM model, using techniques from [38] where instances of *wait-signal* objects are used. Lastly, there is no lower bound for the randomized mutual exclusion problem against the weak adversary that uses only reads-write registers and `CAS` objects. If a lower bound of $\Omega(\log N/\log\log N)$ is established then our algorithm would be optimal against the weak adversary.

Appendix A

Remaining proofs of RandALockArray

Claim A.0.4. *Suppose a process p executes a call to* $\texttt{lock}_p\texttt{()}$ *during a passage. The value of* $\mathsf{Role}[p]$ *at various times is as follows.*

Points in time	*Value of* $\mathsf{Role}[p]$
t_p^5	$\{\infty, \mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN}\}$
$[t_p^7, t_p^8]$	PAWN
t_p^9	$\mathsf{PAWN\_P}$
t_p^{13-}	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$
t_p^{14}	QUEEN
$[t_p^{16}, t_p^{17}]$	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$

Proof. Since the values returned by a $\texttt{Ctr.inc()}$ operation are in $\{\infty, 0, 1, 2\} = \{\infty, \mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN}\}$, $\mathsf{Role}[p]$ is set to one of these values in line 5. Hence, $\mathsf{Role}[p] \in \{\infty, \mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN}\}$ at t_p^5. If p satisfies the if-condition of line 6, then $\mathsf{Role}[p] = \mathsf{PAWN}$, and p changes $\mathsf{Role}[p]$ next only in line 9. Hence, $\mathsf{Role}[p] = \mathsf{PAWN}$ during $[t_p^7, t_p^8]$. In line 9 p changes $\mathsf{Role}[p]$ to $\mathsf{PAWN\_P}$ and does not change $\mathsf{Role}[p]$ thereafter. Hence, $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ at t_p^9.

Process p does not change $\mathsf{Role}[p]$ after line 9. To break out of the getLock loop, $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ must be satisfied when p executes line 12. Hence, $\mathsf{Role}[p] = \{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ during $[t_p^{16}, t_p^{17}]$. Since p executes line 13 only after breaking out of the getLock loop, $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ at t_p^{13-}. If p satisfies the if-condition of line 13, then $\mathsf{Role}[p] = \mathsf{QUEEN}$, and since p does not change $\mathsf{Role}[p]$ thereafter, $\mathsf{Role}[p] = \mathsf{QUEEN}$ at t_p^{14}. ☐

Claim A.0.5. *Suppose a process p executes a call to* $\texttt{abort}_p\texttt{()}$. *The value of* $\mathsf{Role}[p]$ *at* *various points in time is as follows.*

Points in time	*Value of* $\mathsf{Role}[p]$
$[t_p^{19}, t_p^{20-}]$	$\{\mathsf{QUEEN}, \mathsf{PAWN}\}$
t_p^{21}	PAWN
$[t_p^{22}, t_p^{23}]$	$\mathsf{PAWN\_P}$
$[t_p^{26-}, t_p^{30}]$	QUEEN

Proof. Process p calls `abort`$_p$`()` only if p has received a signal to abort and p is busy waiting in one of lines 2, 7, or 14. Then, the last line executed by p before calling `abort`$_p$`()` is line 2, 7, or line 14. From Claim A.0.4, it follows that $\mathsf{Role}[p] = \mathsf{PAWN}$ at t_p^7, and $\mathsf{Role}[p] = \mathsf{QUEEN}$ at t_p^{14}.

Now, p's local variable $flag$ is set to value `true` for the first time in line 3. If p fails the if-condition of line 18, then p must have executed line 3, and thus p broke out of the busy-wait loop of line 2. Then, p last executed line 7 or line 14 before calling `abort`$_p$`()`. Hence, $\mathsf{Role}[p] \in \{\mathsf{PAWN}, \mathsf{QUEEN}\}$ in $[t_p^{19}, t_p^{20}]$, since p changes $\mathsf{Role}[p]$ next only in line 22.

If p satisfies the if-condition of line 20, then $\mathsf{Role}[p] = \mathsf{PAWN}$, and p changes $\mathsf{Role}[p]$ next only in line 22. Hence, $\mathsf{Role}[p] = \mathsf{PAWN}$ at t_p^{21}. In line 22 p changes $\mathsf{Role}[p]$ to $\mathsf{PAWN\_P}$ and p does not change $\mathsf{Role}[p]$ after that. Hence, $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ during $[t_p^{22}, t_p^{23}]$. If p does not satisfy the if-condition of line 20, then $\mathsf{Role}[p] = \mathsf{QUEEN}$ at $[t_p^{26-}, t_p^{30}]$ follows. $\square$

Claim A.0.6. *Suppose a process p executes a call to* `release`$_p$`(`j`)` *during a passage. The value of* $\mathsf{Role}[p]$ *at various points in time is as follows.*

Points in time	*Value of* $\mathsf{Role}[p]$
$[t_p^{34-}, t_p^{35-}]$	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$
$[t_p^{36-}, t_p^{39}]$	KING
t_p^{43-}	QUEEN
t_p^{46-}	$\mathsf{PAWN\_P}$
$[t_p^{49-}, t_p^{50}]$	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$

Proof. Suppose the point in time t_p^{34-}. Then, p is is executing a call to `release`$_p$(j), and p last executed a call to `lock`$_p$`()` that returned a non-$\perp$ value. Then, p's call to `lock`$_p$`()` either returned from line 17 in `lock`$_p$`()` or from line 23 or line 27 in `abort`$_p$`()`. From Claim A.0.4, $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ at time t_p^{17-} and from Claim A.0.5, $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ at t_p^{23-} and $\mathsf{Role}[p] = \mathsf{QUEEN}$ at t_p^{27-}. Therefore, $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ at time t_p^{34-}.

From Claim 4.4.2(b), $\mathsf{Role}[p]$ is unchanged during `release`$_p$`()`. Therefore, $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ during $[t_p^{34-}, t_p^{35-}]$ and $[t_p^{49-}, t_p^{50}]$. Then, from the if-conditions of lines 35, 42 and 45, it follows immediately that $\mathsf{Role}[p] = \mathsf{KING}$ during $[t_p^{36-}, t_p^{39}]$, and $\mathsf{Role}[p] = \mathsf{QUEEN}$ at t_p^{43-}, and $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ at t_p^{46-}.

□

Claim A.0.7. *Suppose a process p executes a call to* `doCollect`$_p$`()`*,* `helpRelease`$_p$`()` *or* `doPromote`$_p$`()` *during a passage. The value of* $\mathsf{Role}[p]$ *at various points in time is as follows.*

Points in time	*Value of* $\mathsf{Role}[p]$
$[t_p^{51-}, t_p^{55}]$	$\{\mathsf{KING}, \mathsf{QUEEN}\}$
$[t_p^{56-}, t_p^{63}]$	$\{\mathsf{KING}, \mathsf{QUEEN}\}$
$[t_p^{65-}, t_p^{71}]$	$\{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$

Proof. From the code structure, p does not change $\mathsf{Role}[p]$ during `doPromote()`, `doCollect`$_p$`()` and `helpRelease()`.

From a code inspection, $\texttt{doCollect}_p\texttt{()}$ is called by p only in lines 29, and 38. From Claim A.0.5, $\mathsf{Role}[p] = \mathsf{QUEEN}$ at t_p^{29-} and from Claim A.0.6, $\mathsf{Role}[p] = \mathsf{KING}$ at t_p^{38-}. Since $\mathsf{Role}[p]$ is unchanged during $\texttt{doCollect}_p\texttt{()}$, it follows that $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}\}$ during $[t_p^{51-}, t_p^{55}]$.

Now, suppose p executes a call $\texttt{helpRelease}_p\texttt{()}$. From a code inspection, $\texttt{helpRelease}_p\texttt{()}$ is called by p only in lines 30, 39 and 43. From Claim A.0.5, $\mathsf{Role}[p] = \mathsf{QUEEN}$ at t_p^{30-} and from Claim A.0.6, $\mathsf{Role}[p] = \mathsf{KING}$ at t_p^{39-} and $\mathsf{Role}[p] = \mathsf{QUEEN}$ at t_p^{43-} . Since $\mathsf{Role}[p]$ is unchanged during $\texttt{helpRelease()}$, it follows that $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}\}$ during $[t_p^{56-}, t_p^{63}]$.

Now, suppose p executes a call $\texttt{doPromote}_p\texttt{()}$. From a code inspection, $\texttt{doPromote}_p\texttt{()}$ is called by p only in lines 46 and 62. From Claim A.0.6, $\mathsf{Role}[p] = \mathsf{PAWN\_P}$ at t_p^{46-} and from earlier in this claim, $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}\}$ at t_p^{62-}. Since $\mathsf{Role}[p]$ is unchanged during $\texttt{doPromote()}$, it follows that $\mathsf{Role}[p] \in \{\mathsf{KING}, \mathsf{QUEEN}, \mathsf{PAWN\_P}\}$ during $[t_p^{65-}, t_p^{71}]$.

□

Bibliography

[1] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8, September 1965.

[2] H. Attiya, D. Hendler, and P. Woelfel. Tight rmr lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, New York, NY, USA, 2008.

[3] J.H. Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9, 1995. 10.1007/BF01784242.

[4] D. Hendler and P. Woelfel. Randomized mutual exclusion with sub-logarithmic rmr-complexity. *Distributed Computing*, 24(1), 2011.

[5] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3), 2003.

[6] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing.*

[7] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, New York, NY, USA, 2003.

[8] M. Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, 1986.

[9] D. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, August 1998.

[10] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel Distributed Systems*, 1, January 1990.

[11] J. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9, February 1991.

[12] J. H. Anderson and Y.J. Kim. Fast and scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Computing*, 1999.

[13] Y.J. Kim and J.H. Anderson. Adaptive mutual exclusion with local spinning. *Distributed Computing*, 19, January 2007.

[14] J.H. Anderson and Y.J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15, December 2002.

[15] R. Danek and W. Golab. Closing the complexity gap between mutual exclusion and fcfs mutual exclusion. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, New York, NY, USA, 2008.

[16] Y.J. Kim and James H. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, London, UK, UK, 2001.

[17] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *IEEE Real-Time Systems Symposium*, 1992.

[18] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel Distributed Systems*, 1, January 1990.

[19] J.H. Anderson, Y.J. Kim, and T Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16, September 2003.

[20] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12, July 1990.

[21] Inc. SPARC International. *The SPARC architecture manual (version 9)*. Prentice-Hall, 1994.

[22] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture Revision 2.1*. October 2002.

[23] H. Le, J. M. Tendler, S. Dodson, S. Fields, and B. Sinharoy. *IBM e-server PO WER System Microarchitecture*. IBM, October 2001.

[24] MIPS Computer Systems. *MIPS64 Architecture for Programmers, Volume H: The MIPS6 Instruction Set*. August 2002.

[25] R. Site. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, 1992.

[26] P. Jayanti and S. Petrovic. Efficient and practical constructions of ll/sc variables. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, New York, NY, USA, 2003.

[27] W. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. Constant-rmr implementations of cas and other synchronization primitives using read and write operations. In *Proceedings of the twenty-sixth annual symposium on Principles of distributed computing*.

[28] W. Golab. *Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations*. PhD thesis, University of Toronto, 2010.

[29] W. Golab, L. Higham, and P. Woelfel. Linearizable implementations do not suffice for randomized distributed computation. *CoRR*, abs/1103.4690, 2011.

[30] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17, August 1974.

[31] J. H. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3), 1993.

[32] G. Peterson and M. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77. ACM, 1977.

[33] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1), 1987.

[34] J. H. Anderson and Y.J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1), 2001.

[35] Y. J. Kim and J. H. Anderson. Adaptive mutual exclusion with local spinning. *Distributed Computing*, 19(3), 2007.

[36] R. Danek and W. Golab. Closing the complexity gap between fcfs mutual exclusion and mutual exclusion. *Distributed Computing*, 23(2), 2010.

[37] P. Jayanti. f-arrays: implementation and applications. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, New York, NY, USA, 2002.

[38] D. Hendler and P. Woelfel. Adaptive randomized mutual exclusion in sub-logarithmic expected time. In *Proceeding of the twenty-ninth annual symposium on Principles of distributed computing*, PODC '10, New York, NY, USA, 2010.

[39] J. L. W. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17, 1982.

[40] M. Herlihy. A methodology for implementing highly concurrent objects. *ACM Transactions on Programming Languages and Systems*, 15(5), 1993.

[41] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, March 2008.