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

© Abhijeet Pareek 2012
The author of this thesis has granted the University of Calgary a non-exclusive license to reproduce and distribute copies of this thesis to users of the University of Calgary Archives.

Copyright remains with the author.

Theses and dissertations available in the University of Calgary Institutional Repository are solely for the purpose of private study and research. They may not be copied or reproduced, except as permitted by copyright laws, without written authority of the copyright owner. Any commercial use or re-publication is strictly prohibited.

The original Partial Copyright License attesting to these terms and signed by the author of this thesis may be found in the original print version of the thesis, held by the University of Calgary Archives.

Please contact the University of Calgary Archives for further information:
E-mail: uarc@ucalgary.ca
Telephone: (403) 220-7271
Website: http://archives.ucalgary.ca
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].
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

Abstract ................................................................. ii
Acknowledgements ................................................... iii
Table of Contents ....................................................... iv
List of Figures .......................................................... vi

1 Introduction ......................................................... 1

2 Preliminaries ....................................................... 4
2.1 Asynchronous Shared Memory Model .............................. 4
2.2 Shared Memory System Architectures ............................. 8
2.3 Primitive Objects ................................................ 9
2.4 Adversary Models for Randomized Distributed Algorithms ... 11
2.5 Atomicity Versus Linearizability ................................ 13
2.6 The Mutual Exclusion Problem .................................. 14
   2.6.1 The Type Lock ......................................... 14
2.7 Abortable Mutual Exclusion ...................................... 18
   2.7.1 The Type AbortableLock .................................. 18

3 Related Work ....................................................... 21
3.1 Mutual Exclusion Algorithms using only Read-Write Registers 21
   3.1.1 Yang & Anderson’s Lock ................................ 21
3.2 Variants of the Mutual Exclusion Problem ....................... 28
   3.2.1 Fast Mutual Exclusion .................................. 28
   3.2.2 Adaptive Mutual Exclusion .............................. 28
   3.2.3 Abortable Mutual Exclusion - Jayanti’s Abortable Lock .... 29
3.3 Randomized Mutual Exclusion .................................. 35
   3.3.1 Hendler and Woelfel’s Randomized Lock .................. 35
   3.3.2 Adaptive Randomized Mutual Exclusion Lock .......... 41

4 Randomized Abortable Mutual Exclusion ........................ 43
4.1 A Randomized Bounded CAS Counter Object .................... 45
   4.1.1 Type CAScounter and its Sequential Specification ...... 45
   4.1.2 Randomized Linearizable Implementation of Type CAScounter ... 46
   4.1.3 Analysis and Properties of Object RCAScounter ....... 47
4.2 Single-Fast-Multi-Slow Universal Construction ................ 53
   4.2.1 Analysis and Proofs of Correctness .................... 58
4.3 AbortableProArray<sub>k</sub>: An Abortable Promotion Array Type 68
4.4 Array based Randomized Abortable Lock ....................... 71
   4.4.1 Introduction ........................................... 71
   4.4.2 High Level Description ................................... 73
   4.4.3 Implementation / Low Level Description ................ 83
   4.4.4 Analysis and Proofs of Correctness .................... 89
4.5 Tree based Randomized Abortable Lock ......................... 143
   4.5.1 High Level Description .................................. 143
   4.5.2 Implementation / Low Level Description ................ 147
# List of Figures

2.1 Type Lock .......................................................... 14
2.2 An algorithm illustrating the usage of Lock ......................... 15
2.3 Type AbortableLock .................................................. 18
2.4 An algorithm illustrating the usage of AbortableLock .............. 20

3.1 YA2Lock\textsubscript{\textit{n}}: Yang and Anderson’s 2 Process Mutual Exclusion Lock .......... 23
3.2 YALock: Yang and Anderson’s \textit{N} Process Mutual Exclusion Lock .......... 26
3.3 JayantiALock: Jayanti’s Abortable Mutual Exclusion Lock ............ 31
3.4 HWLock: Hendler and Woelfel’s Randomized Mutual Exclusion Lock ........ 36
3.5 HWLock: Hendler and Woelfel’s Randomized Mutual Exclusion Lock (continued) ........ 37

4.1 Sequential Specification of Type CAScounter\textsubscript{k} .............................................. 46
4.2 Implementation of Object RCAScounter\textsubscript{k} ....................................................... 48
4.3 Implementation of Object SFMSUnivConstWeak(T) ....................... 56
4.4 Sequential Specification of Type AbortableProArray\textsubscript{k} ................... 69
4.5 Implementation of Object RandALockArray .............................................. 74
4.6 Implementation of Object RandALockArray (continued) .............. 75
4.7 Legend for Figures 4.8 to 4.16 .............................................. 94
4.8 K’s call to \texttt{lock}(K) ............................................. 107
4.9 K’s call to \texttt{release}(K) ............................................. 111
4.10 Q’s call to \texttt{lock}\textsubscript{\textit{Q}}(\textit{j}) ............................................. 111
4.11 Q’s call to \texttt{release}\textsubscript{\textit{Q}}(\textit{j}) ............................................. 112
4.12 A’s call to \texttt{helpRelease}\textsubscript{\textit{A}}(\textit{j}) ................. 115
4.13 B’s call to \texttt{helpReleaseB}(\textit{j}) ............................................. 116
4.14 B’s call to \texttt{doPromote}(\textit{j}) ............................................. 116
4.15 P\textsubscript{i}’s call to \texttt{lock}\textsubscript{P\textsubscript{i}}(\textit{i}) ............................................. 124
4.16 P\textsubscript{i}’s call to \texttt{release}\textsubscript{P\textsubscript{i}}(\textit{i}) ............................................. 124
4.17 Implementation of Object RandALockTree ............................................. 145
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 [3]. 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 \( O(\log N) \) 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 $O(\log N)$ 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 $O(\log N/ \log \log N)$ 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 $O(\log N/ \log \log N)$ 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 $T = (S, s_0, \delta, O, R)$ where $S$ is a set of states, $s_0 \in S$ is the initial state, $O$ is a set of operations, $R$ is the set of responses, and $\delta : S \times O \rightarrow S \times R$ is a many-to-one mapping. The mapping $\delta$ defines how operations are applied on object $J$. Specifically, if a process applies operation $op \in O$ to an object $J$ of type $T$ 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 $J$ of base objects of specified types, we can implement an object of a target type $T$. An *implementation* defines how to simulate an object of type $T$, using base objects $J$. Formally, an implementation of an object of type $T$ is denoted by the tuple $I = (P, J, H, T)$ where $P$ is the set of processes, $J$ is the set of base objects, $H$ is the set of histories, $T$ is the target object type. The set $H$ 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 $J$.

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 $I = (T, P, J, H)$ is linearizable, if every history $H \in H$ can be linearized to a sequential history in the sequential specification of $T$.

**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 $R.\text{Read}()$ and $R.\text{Write}()$. Operation $R.\text{Read}()$ returns the value of the register and leaves its content unchanged, and operation $R.\text{Write}(v)$ 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 $O.\text{LL}()$ and $O.\text{SC}()$. Operation $O.\text{LL}()$ returns the value stored in $O$. Operation $O.\text{SC}(v)$ by a process $p$ must follow the execution of an $O.\text{LL}()$ operation by $p$, and the operation may succeed or fail. A successful $O.\text{SC}(v)$ changes the value of $O$ to $v$ and returns true, otherwise the value of $O$ is unchanged and false is returned. Operation $O.\text{SC}(v)$ 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 $O.\text{CAS}()$ and $O.\text{Read}()$. Operation $O.\text{Read}()$ returns the value stored in $O$. Operation $O.\text{CAS}(exp, new)$ 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 $\text{true}$ is returned. Otherwise, the operation fails, and the value of $O$ remains unchanged and $\text{false}$ 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 $O(1)$-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 $\rho_i$ 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 [20].

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

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 $H_{M,A,c}$ is generated, such that all processes perform steps as dictated by $M$, and $H_{M,A,c}$ observes the coin flip vector $\vec{c}$ and the schedule $A(\vec{c})$.

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

**Definition 2.4.3.** Adversary $A$ is weak for algorithm $M$ if it is adaptive for algorithm $M$ and is additionally constrained so that, in $H_{M,A,c}$, 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 $O(1)$ 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.

<table>
<thead>
<tr>
<th>Type Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods:</td>
</tr>
<tr>
<td>lock()</td>
</tr>
<tr>
<td>release()</td>
</tr>
</tbody>
</table>

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 \texttt{Lock} must satisfy the following:

\textbf{Condition 2.6.1.} A process calls method \texttt{release()} if and only if its last call on the lock object was a completed \texttt{lock()} call.

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

\begin{verbatim}
Algorithm 2: Mutual Exclusion Algorithm Template for Process \textit{p}

// shared: L: An instance of an object of type Lock
while true do
  <Remainder Section>
  L.lock() // Entry Section
  <Critical Section>
  L.release() // Exit Section
end
\end{verbatim}

Figure 2.2: An algorithm illustrating the usage of \texttt{Lock}.

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

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

\textbf{Deadlock Freedom:} If all processes in the system take enough steps, then at least one of them will return from its \texttt{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 \textit{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 \textit{progress} property. There are other properties that are often desirable, such as \textit{starvation freedom} and \textit{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 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. `await(X = 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-⊥ value if the process calling lock() successfully captures the lock, otherwise method lock() returns the special value ⊥. Method lock() returns a ⊥ 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-⊥ 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.

<table>
<thead>
<tr>
<th>Type AbortableLock</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods:</td>
</tr>
<tr>
<td>lock() */ returns ⊥ only if a process receives a signal */</td>
</tr>
<tr>
<td>release()</td>
</tr>
<tr>
<td>*/ to abort during the lock attempt */</td>
</tr>
</tbody>
</table>

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 \texttt{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 \texttt{lock()} call, it does not execute the Critical Section or Exit Section, but returns to the Remainder Section. In the context of type \texttt{AbortableLock}, we redefine a \textit{passage}. If a process' \texttt{lock()} call returns \texttt{⊥}, then the process's passage is defined to be its Entry Section. If a process' \texttt{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 \texttt{lock()} call. We define an \textit{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 \textit{ bounded abort}, and is stated as follows:

\textbf{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 \texttt{AbortableLock} must satisfy the following:

\textbf{Condition 2.7.1}. A process calls method \texttt{release()} if and only if its last call on the lock object was a successful \texttt{lock()} call.

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

\textbf{Specification 2.7.1}. An execution of an algorithm that accesses an instance of an object of type \texttt{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() ≠ ⊥ 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 $O(\log N)$ 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 YA2Lock (see Figure 3.1). We first
describe object YA2Lock and later show how the same idea as that of YA2Lock is used in
an arbitration tree to implement the N-process lock YALock (see Figure 3.2).

Complete Description of 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, $\text{Want}[0]$, $\text{Want}[1]$, $\text{Victim}$, $\text{Spin}[p]$, and $\text{Spin}[q]$. Note that
process $p$ can access $\text{Spin}[q]$ only after determining the ID of process $q$ (and vice versa).
Register $\text{Want}[i]$, for $i \in \{0, 1\}$, ranges over values in $\{p, q, \bot\}$, and is used by a process
to inform the other of its intent to capture the lock. Observe that $\text{Want}[0] = p = \bot$ holds
while process $p$ is at lines 2-14, and $\text{Want}[p] = \bot$ holds otherwise. Similarly for $\text{Want}[1]$. 
Class YA2Lock

shared:
  Want[0], Want[1]: int init ⊥
  Spin[p], Spin[q]: int init ⊥
  Victim: int init ⊥
local:
  rival: int init ⊥
  side: int init ⊥

Method lock()

1. side ← p mod 2
2. Want[side] ← p
3. Victim ← p
4. Spin[p] ← 0
5. rival := Want[1 - side]
6. if rival = ⊥ ∧ Victim = p then
    7.    if Spin[rival] = 0 then
    8.       Spin[rival] ← 1
    9.    end
   10.  await Spin[p] ≥ 1
11.  if Victim = p then
    12.     await Spin[p] = 2
   13.  end
14. end

Method release()

15. Want[side] ← ⊥
16. rival ← Victim[node]
17. if rival = p then
18.    Spin[rival] ← 2
19. end

Figure 3.1: YA2Lockn: Yang and Anderson’s 2 Process Mutual Exclusion Lock
Register $\text{Want}[q]$ is used similarly. Register $\text{Victim}$ 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 $\text{Victim}$ is written to in line 3, determines the order in which $p$ and $q$ capture the lock, in case of contention. Register $\text{Spin}[i]$, 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 $\text{Want}[0]$ (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 $\text{Victim}$ (in line 3), and initializes its spin location $\text{Spin}[p]$ to initial value 0 (in line 4). Then $p$ reads $\text{Want}[1]$ to determine $q$’s ID in case $q$ is contending for the lock too ($q$ writes its ID to $\text{Want}[1]$ in line 2). If $p$ determines that $q$ is not contending for the lock, i.e., if $\text{Want}[1] = \bot$ holds when $p$ executes line 5, then process $p$’s if-condition in line 6 fails, and $p$ proceeds to return from its $\text{lock}()$ call, thus capturing the lock. Otherwise, $p$ reads the tie-breaker variable $\text{Victim}$ in line 6. If $\text{Victim} = p$, which implies that $\text{Victim} = q$, then $p$ can proceed to capture the lock, since $p$ wrote to $\text{Victim}$ first. The algorithm will prohibit $q$ from capturing the lock by the same path as long as $\text{Want}[0] = p \land \text{Victim} = q$ holds ($p$ resets $\text{Want}[0]$ only in line 15). If $\text{Victim} = p$ 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 $\text{Victim}$ again, or for $q$ to release the lock ($q$ notifies $p$ by writing 2 to $\text{Spin}[p]$ 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 $\text{Victim}$, in case $q$ is busy waiting in line 10, waiting for $p$ to update the
tie-breaker. Process \( p \) then busy-waits on \( \text{Spin}[p] \) in line 10, waiting for a notification from \( q \) to indicate that \( q \) has indeed written to \( \text{Victim} \). When \( p \) reads \( \text{Spin}[p] \geq 1 \), 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 \( \text{Victim} \) in line 11 to determine who wrote to \( \text{Victim} \) last. In case \( p \) wrote to \( \text{Victim} \) last, \( p \) busy-waits on \( \text{Spin}[p] \) in line 12, awaiting a notification from \( q \) that would indicate that \( q \) has released the lock.

During the \texttt{release()} method call, process \( p \) first resets \( \text{Want}[0] \) 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 \texttt{YALock} is shown in Figure \ref{fig:YALock}, where the processes are assumed to have unique IDs in the set \( \{0, \ldots, N-1\} \). Object \texttt{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 \texttt{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 \texttt{Want} variable associated with the node.

Method \texttt{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 \texttt{lock()} call, and therefore can enter its Critical Section.
Class YALock

shared:
  \( L \): height of the root node.
  \( \text{Want}[1 \ldots N - 1][0 \ldots 1] \): \textbf{int init} \perp
  \( \text{Victim}[1 \ldots N - 1] \): \textbf{int init} \perp
  \( \text{Spin}[1 \ldots L][0 \ldots N - 1] \): \textbf{int init} \perp

local:
  \( \text{rival}, h, node, side \): \textbf{int init} 0

Method lock()

20 \textbf{for} \ h \leftarrow 1 \ \textbf{to} \ L \ \textbf{do}
21 \quad \text{node} \leftarrow \lfloor (N + p)/2^h \rfloor
22 \quad \text{side} \leftarrow \lfloor (N + p)/2^{h-1} \rfloor \mod 2
23 \quad \text{Want}[\text{node}][\text{side}] \leftarrow p
24 \quad \text{Victim}[\text{node}] \leftarrow p
25 \quad \text{Spin}[h][p] \leftarrow 0
26 \quad \text{rival} \leftarrow \text{Want}[\text{node}][1 - \text{side}]
27 \quad \textbf{if} \ \text{rival} = \perp \land \text{Victim}[\text{node}] = p \ \textbf{then}
28 \quad \quad \textbf{if} \ \text{Spin}[h][\text{rival}] = 0 \ \textbf{then}
29 \quad \quad \quad \text{Spin}[h][\text{rival}] \leftarrow 1
30 \quad \quad \textbf{end}
31 \quad \quad \textbf{await} \ \text{Spin}[h][p] \geq 1
32 \quad \quad \textbf{if} \ \text{Victim}[\text{node}] = p \ \textbf{then}
33 \quad \quad \quad \textbf{await} \ \text{Spin}[h][p] = 2
34 \quad \quad \textbf{end}
35 \quad \textbf{end}
36 \textbf{end}

Method release()

37 \textbf{for} \ h \leftarrow L \ \textbf{down to} \ 1 \ \textbf{do}
38 \quad \text{node} \leftarrow \lfloor (N + p)/2^h \rfloor
39 \quad \text{side} \leftarrow \lfloor (N + p)/2^{h-1} \rfloor \mod 2
40 \quad \text{Want}[\text{node}][\text{side}] \leftarrow \perp
41 \quad \text{rival} \leftarrow \text{Victim}[\text{node}]
42 \quad \textbf{if} \ \text{rival} = p \ \textbf{then}
43 \quad \quad \text{Spin}[h][\text{rival}] \leftarrow 2
44 \quad \textbf{end}
45 \textbf{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 $O(\log N)$ 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 \( O(\min(k, \log N)) \) RMR time complexity, where \( k \) is the point contention.

**Theorem 3.2.3** ([35]). The mutual exclusion problem can be solved with \( O(\min(k, \log N)) \) 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 \( O(\min(k, \log N)) \) 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 \( O(\min(k, \log N)) \) 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 \( O(\min(k, \log N)) \), 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 \( O(N) \) memory words that support LL, SC, Read and Write operations.
We now provide a high-level description of object \textit{JayantiALock} (see Figure 3.4). Object \textit{JayantiALock} makes use of shared registers that support LL, SC, \texttt{Read} and \texttt{Write} operations. Such a construction is equivalent to a construction that uses only \texttt{CAS}, \texttt{Read} and \texttt{Write} operations, since \texttt{CAS} can be implemented using LL/SC in a wait-free manner in $O(1)$ steps with $O(1)$ space, and vice versa [26]. To recall the specification of the LL/SC operations refer to Section 2.3.

Instance \texttt{Ctr} is an instance of a linearizable counter object \cite{37} that provides methods \texttt{Read()} and \texttt{inc()}, where \texttt{Read} returns the value of \texttt{Ctr} and \texttt{inc}(d) increments the value of \texttt{Ctr} by $d$. Instance \texttt{Q} is an instance of a linearizable priority process-queue object \cite{37} that provides methods \texttt{insert()}, \texttt{findmin()} and \texttt{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 \texttt{f-array} objects \cite{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.

\textbf{Theorem 3.2.6} (\cite{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 \texttt{Read()} is $O(1)$ and the time complexity of \texttt{inc}(d) is $O(\min(k, \log N))$, where $k$ is the point-contention during the increment operation. The space complexity is $O(N)$.

\textbf{Theorem 3.2.7} (\cite{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 \texttt{findmin()} is $O(1)$ and of a matching pair of operations - \texttt{insert()} followed by \texttt{delete()} – is $O(\min(k, \log N))$, where $k$ is point-contention during the execution of \texttt{insert()}. The space complexity is $O(N)$. 

### Class JayantiALock

**shared:**

- Ctr is a counter, initially 0; supports `inc()` and `Read()` operations.
- Q is a priority process-queue, initially empty; supports `insert()`, `findmin()` and `delete()` operations.
- CSowner takes on a value from \{⊥, 0, 1, ..., N − 1\}, initially ⊥; supports LL, SC, Read and Write operations.
- Wait is an array [0 ... N − 1] of boolean initially ⊥; supports LL, SC, Read and Write operations.

**local:**

- \( t, t', q : \text{int init} \perp \)
  // 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. \( \text{Wait}[i] \leftarrow \text{true} \)
2. \( \text{Ctr}.inc(I) \)
3. \( t \leftarrow \text{Ctr}.Read() \)
4. \( Q.\text{insert}((i, t)) \)
5. \( \text{promote()} \)
6. \( \text{promote()} \)
7. \( \text{await}(\neg \text{Wait}[i]) \)
8. \( \text{return } i \)

#### Method `release_i()`

9. \( Q.\text{delete}((i, t)) \)
10. \( \text{CSowner} \leftarrow \perp \)
11. \( \text{promote()} \)

#### Method `promote()`

19. \( \text{if CSownerLL()} = \perp \text{ then return } \)
20. \( (j, t') \leftarrow Q.\text{findmin}() \)
21. \( \text{if } j = \perp \text{ then Wait}[j].\text{LL}() \)
22. \( \text{if CSownerSC}(j) \text{ then } \)
23. \( \text{if } j = \perp \text{ then Wait}[j].\text{SC}(\text{false}) \)
24. \( \text{end } \)

**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 $\text{Wait}[p]$ to true at the start of $\text{lock()}$, and when the owner of the lock makes $p$ the owner, it releases $p$ from its busy-wait loop by writing false to $\text{Wait}[p]$. An LL/SC register $\text{CSowner}$ holds the name of the process that owns $\text{JayantiALock}$ If no process currently owns $\text{JayantiALock}$ then $\text{CSowner} = \bot$.

Object $\text{Ctr}$ is used by processes to receive a token number and this helps in achieving the FCFS property. Specifically, processes first increase $\text{Ctr}$ by 1, and then read $\text{Ctr}$ 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 $\text{Q}$ 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 $(p, t) < (q, t')$ if $t < t'$ or $((t = t') \land (p < q))$.

In the $\text{lock()}$ method, right after receiving its token $t$, a process $p$ inserts $(p, t)$ into $\text{Q}$ using the $\text{insert()}$ method. Method $\text{findmin()}$ 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 $\text{lock()}$ method. First, process $p$ announces its desire to own the lock by setting its flag $\text{Wait}[p]$ to true (line 1). In lines 2 and 3, $p$ increases the counter $\text{Ctr}$ by 1, and then reads $\text{Ctr}$ to obtain a token $t$. Operations on $\text{Ctr}$ are wait-free and $p$ executes $O(\min(k, \log N))$ steps during $\text{Ctr.inc(1)}$ and $O(1)$ during $\text{Ctr.Read()}$, where $k$ is the point contention. The $\text{Ctr}$ object is implemented using a tree structure, with the
root storing the Ctr value, where $O(\min(k, \log N))$ 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 $Q$ to enqueue itself in line 4 using a $Q.insert(\langle p, t \rangle)$ operation, and lines 1-4 constitute the bounded doorway. The insert operation on $Q$ also takes $O(\min(k, \log N))$ steps, since $Q$ is internally represented with a similar tree structure, where the root holds the pair with the lowest lexicographical value, and it takes $O(\min(k, \log N))$ 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 $Q$ 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 $\text{Wait}[p]$ 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 $\text{Wait}[p] = \text{false}$), $p$ returns from its $\text{lock}()$ call in line 8 with a non-$\bot$ value, and enters its Critical Section, and eventually calls $\text{release}()$. During $\text{release}()$, $p$ removes its pair $\langle p, t \rangle$ from the queue $Q$ in line 9 with a $Q.delete(\langle p, t \rangle)$ operation, and resets $\text{COwner}$ to $\bot$ 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 $Q$ the next owner of the lock.

**Description of the abort() method.** While busy-waiting in line 7, process $p$ calls $\text{abort}()$ if $p$ receives a signal to abort. During $\text{abort}()$, $p$ removes its pair $\langle p, t \rangle$ from the queue $Q$ in line 12 with a $Q.delete(\langle p, t \rangle)$ operation, and then executes the promote procedure in line 13 to make the longest waiting process (if any) in $Q$ the next owner of the lock. It may happen that some process promoted $p$ by executing the $\text{promote}()$ procedure before $p$ deletes its entry from $Q$ 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 \( \text{CSowner} = p \), and if so it resets \( \text{CSowner} \) 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 \( Q \) the next owner of the lock. In line 18 \( p \) returns with a \( \perp \) value to indicate that it aborted during its \text{lock()} call.

**Description of the promote() method.** The promote() method advances the longest waiting process in \( Q \) 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 \( Q \) with the smallest lexicographical pair, i.e., the longest waiting process using the \( Q.\text{findmin()} \) operation in line 20. If the queue is not empty, \( \text{findmin()} \) returns a process ID \( q \), otherwise it returns \( q = \perp \). First suppose that \( q = \perp \). Then, \( p \) load-links the value of \( \text{Wait}[q] \) in line 21. Then it attempts to write \( q \) into \( \text{CSowner} \) using an \text{SC} operation \( \text{CSowner.\text{SC}}(q) \) in line 22, and if successful it changes \( \text{Wait}[q] \) to \text{false} using \( \text{Wait}[q].\text{SC}($false$) \) 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 \( \text{CSowner} \) 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 \( \text{findmin()} \) returns \( \perp \), \( p \) attempts to write \( \perp \) into \( \text{CSowner} \) using a \text{SC} operation \( \text{CSowner.\text{SC}}(q) \) in line 22. It seems counter intuitive to write \( \perp \) into \( \text{CSowner} \), 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 $\Delta$. 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 $\Delta$-ary tree of height $\Delta$ 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 $\Delta$ 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 $\mathcal{T}$ 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()

1. notify[p] := false
2. v := leaf[p]
3. repeat
    4. Let i be the integer such that v is the (i + 1)-th child of parent(v)
    5. v := parent(v)
    6. v.apply[i].CAS(⊥, p)
    7. ctr := 0
    8. repeat
        9. ctr := ctr + 1
        10. if ctr > ⌈log Δ⌉ then
            11. if v.apply[i].CAS(p, ⊥) then
                12. v.MX.GetLocki()
                13. v.apply[i].CAS(⊥, p)
                14. await (v.Lock = ⊥ ∨ v.apply[i] = p)
            15. end
        16. end
        17. if ¬v.Lock.CAS(⊥, p) then
            18. tok := v.token
            19. await (v.token = tok ∨ v.apply[i] = p ∨ v.Lock = ⊥)
        20. end
        21. if v.MX.LockOwner = i then v.MX.RelLocki()
        22. until v.apply[i] = p ∨ v.Lock = p
        23. if ¬v.apply[i].CAS(p, ⊥) then
            24. await (notify[p] = true)
        25. end
    26. until notify[p] ∨ v = root

Figure 3.4: HWLock: Hendler and Woelfel’s Randomized Mutual Exclusion Lock
Method release()

27 foreach node v on the path from leaf[p] to the root, where v.Lock = p do
28     tok := v.token
29     i := v.MX.LockOwner
30     Pick j’ uniformly at random from {0, …, Δ − 1}
31     for j ∈ {j’, tok, i} − {⊥} do
32         q := v.apply[j]
33         if q = ⊥ ∧ v.apply[j].CAS(q, ⊥) then
34             promQ.enq(q)
35         end
36     end
37     v.token := (tok + 1) mod Δ
38     if v = root then v.Lock.CAS(p, ⊥)
39 end
40 if promQ = ∅ then
41     root.Lock.CAS(p, ⊥)
42 else
43     q := promQ.deq()
44     root.Lock.CAS(p, q)
45     notify[q] := true
46 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(⊥, 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 \( v.\text{Lock} \), waiting for \( v.\text{Lock} \) to be released. On seeing a release (i.e. reading \( v.\text{Lock} = \bot \)) 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 \( \Delta \). 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 \( \text{lock}() \) is \( \mathcal{O}(\Delta) \). In its Exit Section, \( p \) releases all the locks acquired on its path \( \text{path}_p \) 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 \( \text{apply}[0\ldots\Delta - 1] \). The unique position is the index of the child node of \( v \) from which \( p \) ascended to \( v \). The process that owns the lock \( v.\text{Lock} \), 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 \( v.\text{apply}[i] \) for a registered process. If \( q \) finds some process registered at \( v.\text{apply}[i] \), say \( p' \), \( q \) then attempts to promote \( p' \) by executing a \( v.\text{apply}[i].\text{CAS}(p', \bot) \) 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 \( \text{promQ} \).
Thus, every registered process has a $1/\Delta$ chance to be promoted, during a randomized promotion event at the node.

Promoted processes enqueued into $\text{promQ}$, 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 $\text{promQ}$ (if any), to enter the Critical Section. With this mechanism, all processes enqueued in the queue $\text{promQ}$ are guaranteed to be notified to enter the Critical Section, and all enqueue and dequeue operations on $\text{promQ}$ 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 $\Delta$, a process has to capture at most $\Delta$ 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-$\Delta$ counter $v\.token$ is maintained, and is increased only by the exiting process (and thus, in mu-
tual 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 $j = v.token$, and then attempting to promote the process (if any) at $v.apply[j]$, as described before. Process $q$ then increments $v.token$ (modulo $\Delta$). Such a mechanism guarantees that if $\Delta$ promotion events occur at node $v$ while a process is busy-waiting on $v.Lock$, 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 $O(\Delta)$ (to match the lower bound of deterministic mutual exclusion algorithms), HWLock makes use of a starvation free $\Delta$-process deterministic mutual exclusion object $v.MX$ at every node $v$ of the arbitration tree. The object $v.MX$ provides methods $\text{lock()}$ and $\text{release()}$, which can be called with unique IDs in $\{0, \ldots, \Delta - 1\}$, and has worst-case RMR complexity $O(\log \Delta)$ (lock $YALock$ of [3] can be used as an implementation of $MX$). The algorithm assumes the presence of another method $\text{LockOwner()}$ that returns the ID of the current owner of $v.MX$.

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 $O(\log \Delta)$ RMRs, then it calls $v.MX.lock_i()$, where $i$ is the rank of the unique child from which the process ascended to $v$. Once the process has captured the lock $v.MX$, 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 $\text{lock}()$ and $\text{release}()$. 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 $O(\log N/\log \log N)$, in expectation and $O(\log n)$ in the worst-case, against the adaptive adversary. The algorithm requires $O(N)$ 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 $O(\log k / \log \log k)$ RMRs per passage through the Critical Section. Against an adaptive adversary the expected amortized RMR complexity is also $O(\log k / \log \log k)$ RMRs. The worst-case RMR-complexity of the algorithm is $O(\min(k \log k, \log N))$. 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 $O(\log k / \log \log k)$ and worst-case RMR complexity of $O(\min(k \log k, \log N))$ 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 $O(\log N)$ worst-case RMR complexity per passage, which is optimal for deterministic algorithms. We have also seen a randomized lock, HWLock with $O(\log N/\log \log N)$ 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 $O(\log N/\log \log N)$ 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 $O(1)$ 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 $O(\log N/\log \log N)$ 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 `CAScounter_k` and its Sequential Specification

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

Object \( C \) 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<sub>k</sub>

<table>
<thead>
<tr>
<th>shared:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x: \text{int} \text{ init} 0 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation inc()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if ( x = k ) then return ( x )</td>
</tr>
<tr>
<td>2 ( x \leftarrow x + 1 )</td>
</tr>
<tr>
<td>3 return ( x - 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation CAS(old, new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 if ( x = old \lor new \notin {0, 1, \ldots, k} ) then return false</td>
</tr>
<tr>
<td>5 ( x \leftarrow new )</td>
</tr>
<tr>
<td>6 return true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation Read()</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 return ( x )</td>
</tr>
</tbody>
</table>

Figure 4.1: Sequential Specification of Type CAScounter<sub>k</sub>

4.1.2 Randomized Linearizable Implementation of Type CAScounter<sub>k</sub>

We now implement object RCAScounter<sub>k</sub> which is a randomized linearizable implementation of type CAScounter<sub>k</sub>. The implementation of object RCAScounter<sub>k</sub> 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 ⊥ 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 \( \beta \). 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 ⊥ (in line 14) to indicate a failed inc() method call. If β ∈ {0, . . . , k − 1}, then p performs a Count.CAS(β, β + 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 ⊥ (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, . . . , 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 RCAScounterk

Consider an instance of the RCAScounterk 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<sub>k</sub>

shared:
    Count: int init 0

local:
    β: int init 0

Method inc()

8  β ← random(0,1,...,k)
9  if (β = k) then
10   if (Count.Read() = k) then return k
11  else
12   if Count.CAS(β, β + 1) then return β
13 end
14 return ⊥

Method CAS(old, new)

15 if new ∉ \{0,1,...,k\} then return false
16 return Count.CAS(old, new)

Method Read()

17 return Count.Read()

Figure 4.2: Implementation of Object RCAScounter<sub>k</sub>
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 $\text{RCAScounter}_k$ is a randomized linearizable implementation of the type $\text{CAScounter}_k$, we need to show that the sequential history $S$ is valid, i.e., $S$ lies in the specification of type $\text{CAScounter}_k$ object, and that $pt(u)$ lies in $I(u)$. Let $C$ be an object of type $\text{CAScounter}_k$, and let $S_v$ be the sequential history obtained when the operations of $S$ are executed sequentially on object $C$ in the order as given in $S$. Clearly, $S_v$ is a valid sequential history in the specification of type $\text{CAScounter}_k$ by construction. Then to show that $S$ is valid, we show that $S = S_v$.

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

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

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

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

If $u$ is a $\text{CAS()}$ 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 $\text{CAS}$ 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 $\text{Count}(u_i)^+$ denote the value of object $\text{Count}$ 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 $\text{Read()}$ 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 $\text{Count}(u_i)^+ = \mathcal{C}(v_i)^+$, 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 $\text{Count}$ and the value of the atomic $\text{CAScounter}_k$ object is 0, it follows from the definition of the method call $u_0$, that $\text{Count}(u_0)^+ = \mathcal{C}(v_0)^+ = 0$, and the return value of $u_0$ matches that of $v_0$.

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

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

**Case b** - $u_i$ is an $\text{inc()}$ method call that returns from line $10$. Then $pt(u_i)$ is when the $\text{Read()}$ operation on the object $\text{Count}$ is executed in line $10$. Clearly, the value returned by the $\text{Read()}$ operation on the object $\text{Count}$ at $pt(u_i)$ is $\text{Count}(u_{i-1})^+$. Since the if-condition of line $10$ is satisfied, $\text{Count}(u_{i-1})^+ = k$ and $u_i$ returns integer $k$ without
changing object Count. Now consider operation $v_i$ in $S_v$. Since $C(v_{i-1})^+ = \text{Count}(u_{i-1})^+$ and $\text{Count}(u_{i-1})^+ = k$, the if-condition of line 14 is satisfied and integer $k$ is returned without changing the atomic CAScounter$_k$ object. Hence $\text{Count}(u_i)^+ = C(v_i)^+$ 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 $\text{new} \notin \{0,1,\ldots,k\}$ and $u_i$ returns false without changing Count. Now consider operation $v_i$ in $S_v$. Since $\text{new} \notin \{0,1,\ldots,k\}$, the if-condition of line 4 will be satisfied and the Boolean value false is returned without changing the value of object $C$. Hence $\text{Count}(u_i)^+ = C(v_i)^+$ 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 $\text{Count}(u_{i-1})^+ = \text{old}$ then $\text{Count}(u_i)^+ = \text{new}$ 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 $C(v_{i-1})^+ = \text{old}$ then $C(v_i)^+ = \text{new}$ and the Boolean value true is returned. And if $C(v_{i-1})^+ = \text{old}$ then the value of object $C$ remains unchanged and the Boolean value false is returned. Hence $\text{Count}(u_i)^+ = C(v_i)^+$ and the return values match.

\[\square\]

**Lemma 4.1.2.** The probability that an inc() method call returns ⊥ 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 $\beta$ in line 8 and before $p$ finishes executing its next shared memory operation. From the code structure,
\( p \) returns \( \bot \) 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.

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

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

The following theorem follows from Lemmas 4.1.1-4.1.3.

**Theorem 4.1.1.** *Object RCAScounter}_k *is a randomized wait-free linearizable implementation of type CAScounter}_k *, where the probability that an \( \text{inc()} \) method call fails is \( \frac{k}{k+1} \) against the weak adversary. *Each of the methods of RCAScounter}_k *has step complexity \( O(1) \).*
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 $O(1)$ steps and no operation takes more than $O(n)$ 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 $O(1)$ steps and some other operations executable in $O(n)$ 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 $\text{SFMSUnivConst}(T)$ that implements an object $O$ of type $T$.

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

In this section, rather than implementing object $\text{SFMSUnivConst}(T)$, we implement a lock-free universal construction object $\text{SFMSUnivConstWeak}(T)$, with slightly weaker properties than $\text{SFMSUnivConst}(T)$. 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 $\text{SFMSUnivConstWeak}(T)$ has the same properties as object $\text{SFMSUnivConst}(T)$ except method $\text{performFast}()$ is lock-free with unbounded step-complexity.
There is a standard technique called \emph{operation combining} \cite{Herlihy1990} that can be applied to transform our lock-free object $\text{SFMSUnivConstWeak}(T)$ to the wait-free object $\text{SFMSUnivConst}(T)$ with $O(n)$ step complexity for method \text{performFast}().

\textbf{Operation Combining Technique.} In principle the technique works as follows: Processes maintain an $n$-element array, say \text{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 \text{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 \text{response}, which can sometimes be combined with the \text{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 \text{announce} array, a process needs to help announced operations $O(n)$ times before its own announced operation is applied, at which point it can stop.

Herlihy \cite{Herlihy1990} introduced this technique as a general methodology to transform lock-free universal constructions to wait-free ones. Herlihy presents another example \cite{Herlihy1991} 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 $O(n)$.

By applying the technique of operation combining we can transform our lock-free universal construction $\text{SFMSUnivConstWeak}(T)$ into our wait-free object $\text{SFMSUnivConst}(T)$. 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 $SFMSUnivConstWeak(T)$, and the proofs illustrate the main idea from this section, i.e., how to achieve a linearizable concurrent implementation with support for a $\text{performFast}()$ method of $O(1)$ step complexity. We now present the implementation of object $SFMSUnivConstWeak(T)$ (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 $\bot$ 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 $(\bot, 0)$. The methods $\text{performFast}()$ and $\text{performSlow}()$ make use of two private methods $\text{helpFast}()$ 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 (s₀, ⊥, 0, 0)
  fastOp: int init (⊥, 0)

local:
  state, res, fc, sc, s₁, s₁, r₁, r₂, seq: int init 0

Method performFast(op)

1. \((state, res, fc, sc) \leftarrow mReg.Read()\)
2. fastOp ← (op, fc + 1)
3. if ¬helpFast() then helpFast()
4. \((state, res, fc, sc) \leftarrow mReg.Read()\)
5. return res

Method f(state₁, op)

6. \(state₂ \leftarrow\) state generated when \(op\) is applied to object \(O\) with state \(state₁\)
7. \(res \leftarrow\) result when \(op\) is applied to object \(O\) with state \(state₁\)
8. return (state₂, res)

Method helpFast()

9. \((s₁, r₁, fc, sc) \leftarrow mReg.Read()\)
10. \((op, seq) \leftarrow fastOp.Read()\)
11. if fc ≥ seq then return true
12. \((s₂, r₂) \leftarrow f(s₁, op)\)
13. return mReg.CAS((s₁, r₁, fc, sc), (s₂, r₂, seq, sc))

Method performSlow(op)

14. repeat
15. \((s₁, r₁, fc, sc) \leftarrow mReg.Read()\)
16. \((s₂, r₂) \leftarrow f(s₁, op)\)
17. if s₂ = s₁ then return r₂
18. helpFast()
19. until mReg.CAS((s₁, r₁, fc, sc), (s₂, r₁, fc, sc + 1))
20. 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 $\text{performFast}(op)$. In line 1 process $p$ first copies the 4-tuple read from register $m\text{Reg}$ 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 $\text{helpFast}()$ in line 3. If the call to $\text{helpFast}()$ returns $\text{false}$, then $p$ concludes that the announced operation may not have been performed yet. In this case $p$ makes another call to $\text{helpFast}()$ in line 3 to be sure that the announced operation is performed (we prove later that at most two calls to $\text{helpFast}()$ are required to perform an announced operation). Process $p$ then reads and returns the result of the performed operation stored in $m\text{Reg}[1]$ in line 4 and 5 respectively. Since method $\text{performFast}()$ is not executed concurrently (by assumption), the result of $p$’s operation stored in register $m\text{Reg}$ is not overwritten before the end of $p$’s $\text{performFast}(op)$ call.

**Description of the helpFast() method.** Let $q$ be a process that calls and executes $\text{helpFast}()$. In line 9 process $q$ first copies the 4-tuple read from register $m\text{Reg}$ into its local variables $s1, r1, fc$ and $sc$. The value read from $m\text{Reg}[0]$ constitutes the state of object O, to which $q$ will attempt to apply the announced operation if required. The value read from $m\text{Reg}[1]$ is the result of the last fast operation performed on object O. The value read from $m\text{Reg}[2]$ and $m\text{Reg}[3]$ 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 $\text{true}$, 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 fastOp[1] changes only in line 2.
(b) The value of \( \text{mReg}[3] \) increases by one with every successful \text{CAS} operation in line 19 and no other operation changes \( \text{mReg}[3] \).

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

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

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

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

\( S_1 \) \( \beta_t \) exists and \( \beta_t \) is in \( (\alpha_t, \gamma_t) \)

\( S_2 \) Throughout \( (\alpha_t, \beta_t) \) : \( \text{fastOp}[1] = \text{mReg}[2] + 1 = t \)

\( S_3 \) Throughout \( (\beta_t, \alpha_{t+1}) \) : \( \text{fastOp}[1] = \text{mReg}[2] = t \)
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 $\text{fastOp}[1]$ and $\text{mReg}[2]$ is 0. Consider the interval $(\beta_0, \alpha_1)$. From Claim 4.2.1(a) it follows that $\text{fastOp}$ is written for the first time at $\alpha_1$. The first point when one of the invariants $(S_2)$ is destroyed is if a process (say $p$) executes a successful $\text{CAS}$ operation in line 13 during $(\beta_0, \alpha_1)$. Then $p$ read the value 0 from register $\text{fastOp}[1]$ in line 10, since the initial value of $\text{fastOp}[1]$ is 0 and $\text{fastOp}[1]$ is written to for the first time at $\alpha_1$. Since $\text{mReg}[2]$ is never decremented (from Claim 4.2.1(c)) and $\text{mReg}[2]$ initially has value 0, $p$ satisfies the if-condition of line 11 and $p$’s $\text{helpFast}()$ call returns $\text{true}$ 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 $(\alpha_t, \gamma_t)$. To show that $(S_1)$ holds for $t$, we need to show that $\text{mReg}[2]$ is changed during $(\alpha_t, \gamma_t)$. Consider $p_t$’s first call to $\text{helpFast}()$ in line 3 during $u_t$. From induction hypothesis $(S_3)$ for $t - 1$, it follows that $\text{fastOp}[1] = \text{mReg}[2] = t - 1$ during $(\beta_{t-1}, \alpha_t)$. Then $p_t$ reads value $t - 1$ from $\text{mReg}[2]$ in line 11 and writes value $t$ to $\text{fastOp}[1]$ in line 2. Since $\text{fastOp}[1]$ is changed only at $\alpha_{t + 1}$ after $\alpha_t$, it follows that $p_t$ reads $t$ from register $\text{fastOp}[1]$ in line 10.

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

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

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

Subcase c1 - $q$ changed $\text{mReg}$ by executing line 13: Then $q$ has changed $\text{mReg}[2]$
and hence \((S_1)\) holds for \(t\).

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

If \(p_t\)'s second \(\text{helpFast()}\) call returns \textit{false} from line 13 then some process changed \(m\text{Reg}\) after \(p_t\) read \(m\text{Reg}\) in line 9. If some process changed \(m\text{Reg}\) by executing line 13 then we get that \((S_1)\) holds for \(t\). Then some process changed \(m\text{Reg}\) by executing line 19 after \(p_t\) read \(m\text{Reg}\) in line 9 and let \(r\) be the first process to do so. Therefore, \(r\) changes the value of \(m\text{Reg}[3]\) from \(m\) to \(m + 1\) in line 19. Then \(r\) executed line 15 after \(q\) executed a successful \(\text{CAS}\) operation in line 19. Then \(r\) completed a call to \(\text{helpFast()}\) in line 18 after \(\alpha_t\). Since \(r\) reads \(m\text{Reg}\) after \(\alpha_t\), \(r\) satisfied the if-condition of line 11 and executed line 13. If \(r\) successfully executes the \(\text{CAS}\) operation in line 13 then we get that \((S_1)\) holds for \(t\). Then some process \(s\) must have changed \(m\text{Reg}\) after \(r\) read \(m\text{Reg}\) in line 9. Since the value of \(m\text{Reg}[3]\) is only incremented (by Claim 4.2.1(b)) and \(r\) changes the value of \(m\text{Reg}[3]\) from \(m\) to \(m + 1\), it follows that \(s\) changed \(m\text{Reg}\) 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 \(\beta_t\) exists and \(\alpha_t < \beta_t < \gamma_t < \alpha_{t+1}\). From the induction hypothesis invariants \((S_2)\) and \((S_3)\) are true until \(\alpha_t\). Now, one of the invariants \((S_2)\) or \((S_3)\) can be destroyed only if some process executes a successful \(\text{CAS}\) operation in line 13 and changes \(m\text{Reg}[2]\). By definition of \(\beta_t\), \(m\text{Reg}[2]\) is unchanged during \((\alpha_t, \beta_t)\). Then invariants \((S_2)\) and \((S_3)\) continue to hold until \(\beta_t\). 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 \(\text{CAS}\) operation in line 13 and changes \(m\text{Reg}[2]\) at \(\beta_t\). Since \(m\text{Reg}[2] = t - 1\) immediately before \(\beta_t\) and \(p\) executes a successful \(\text{CAS}\) operation in line 13 at \(\beta_t\), \(p.f.c = t - 1\). Then \(p\) executed lines 9 and 10 during \((\alpha_t, \beta_t)\) and \(p.seq = t\). Therefore, invariant \((S_3)\) is true immediately after \(\beta_t\).
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 $(\beta_t, \alpha_{t+1})$. Then $q$ must have read register $mReg[2]$ and $fastOp[1]$ after $\beta_t$, 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 $\alpha_{t+1}$, 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 \( SFMSUnivConstWeak(T) \) 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 \( mReg[0] \) 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 \( \alpha_t \) and \( \beta_t \) denote the value returned by \( u_t \) and \( v_t \), respectively. Define \( UC_0^+ = UC_1^- \) and \( O_0^+ = O_1^- \). Define \( \alpha_0 = \beta_0 = \bot \).

**Claim 4.2.4.** Suppose a process calls method \( f(x_1, op) \) and the method returns the value pair \((x_2, y)\). If \( x_1 = O_t^- \) then \( x_2 = O_t^+ \) and \( y = \beta_t \).

**Proof.** By definition, a call to method \( f(x_1, op) \) 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 \( x_1 = O_t^- \) then \( x_2 = O_t^+ \) and \( y = \beta_t \). \( \square \)

**Claim 4.2.5.** For all \( t \geq 1 \).

\[
\begin{align*}
(S_1) & \quad O_t^+ = O_{t+1}^- \text{ and } UC_t^+ = UC_{t+1}^- \\
(S_2) & \quad UC_t^- = O_t^-
\end{align*}
\]
\((S_3)\) \(\text{UC}^+_{t-1} = O^+_{t-1}\) and \(\alpha_{t-1} = \beta_{t-1}\)

**Proof. Proof of \((S_1)\):** Since operations in \(S_v\) are executed sequentially, it follows that \(O^+_t = O^-_{t+1}\). We now show that \(\text{UC}^+_t = \text{UC}^-_{t+1}\). Assume \(\text{UC}^+_t = \text{UC}^-_{t+1}\). Then some process \(p\) changed the value of \(\text{mReg}[0]\) by executing a successful \(\text{CAS}\) 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 \(\text{CAS}\) operation in line 13 or line 19 is \(pt(u_\ell)\) for some method call \(u_\ell\) where \(u_\ell\) is the \(\ell\)-th method call in \(H'\). Thus, \(\ell\) 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, \(\text{mReg}[0]\) is the initial state of \(O\), hence, \(\text{UC}^-_1 = O^-_1\). 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, \(O^+_t = O^-_{t+1}\) and \(\text{UC}^+_t = \text{UC}^-_{t+1}\). From \((S_3)\) for \(t\) we have \(\text{UC}^+_t = O^+_t\). Therefore, it follows that \(\text{UC}^-_{t+1} = O^-_{t+1}\) and thus \((S_2)\) for \(t + 1\) is true.

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

**Case a -** \(u_t\) is a \(\text{performSlow}(op)\) method call: Let \(x_1\) be the most recent value read by \(p_t\) from \(\text{mReg}[0]\) 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 \(\text{Read}\) operation on register \(\text{mReg}\) in line 15. Since \(p\) satisfies the if-condition of line 17, \(x_2 = x_1\). Thus, \(\text{UC}^-_t = \text{UC}^+_t = x_1\).

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

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

**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 $mReg[0]$ 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 $UC^-_t = x_1$ and $UC^+_t = x_2$. Since $x_1 = UC^-_t = O^-_t$, from Claim 4.2.4 it follows that $x_2 = O^+_t$ and $y = \beta_{u_t}$. Since $x_2 = UC^+_t$, it follows that $O^+_t = UC^+_t$.

From Claim 4.2.2 it follows that $mReg[1]$ 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 $mReg[1]$ in line 11 since $p$ executes line 11 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 $SFMSUnivConstWeak(T)$ 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 \texttt{performFast()} method calls. Consider a process \( p \) that takes steps in history \( H \) and fails to complete its \texttt{performSlow()} method call. Then during \( p \)'s execution \( p \) reads register \texttt{mReg} in line 15 and fails its \texttt{CAS} operation in line 19 during an iteration of the loop of lines 14-19. Now \( p \)'s \texttt{CAS} operation can fail only if some process executes a successful \texttt{CAS} operation in line 13 or line 19 between \( p \)'s \texttt{Read()} and \texttt{CAS} operation.

**Case a** - Some process \( q \) executes a successful \texttt{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 \texttt{performSlow()} method call – a contradiction.

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

The following theorem follows from Lemma 4.2.1 and 4.2.2.

**Theorem 4.2.1.** Object \( \text{SFMSUnivConstWeak}(T) \) 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 \( \text{SFMSUnivConstWeak}(T) \) concurrently and operations on object \( O \) are performed using either method \texttt{performFast()} or
performSlow(), and no two processes execute method performFast() concurrently. Method performFast() has $O(1)$ step complexity.

On applying the standard technique of operation combining [10] to object SFMSUnivConstWeak(T) we obtain object SFMSUnivConst(T). The following theorem summarizes the properties of object SFMSUnivConst(T).

**Theorem 4.2.2.** Object SFMSUnivConst(T) 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 SFMSUnivConst(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 $O(1)$ step complexity and method performSlow() has $O(n)$ step complexity.
4.3 **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 SFMSUnivConst() is used to get a concurrent wait-free linearizable implementation of such an object. An object \(O\) of type **AbortableProArray\(_k\)** stores a vector of \(k\) integer pairs. Initially the value of \(O\) is \((\langle 0, \bot \rangle, \ldots, \langle 0, \bot \rangle)\). 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 **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 \(\{\bot\} \cup \mathbb{Z}\). For all \(i\) in \(\{0, \ldots, k - 1\}\), if \(O[i] = \langle ABORT, s \rangle\) and \(X[i] = \bot\) then the operation changes \(O[i]\) to value \(\langle REG, X[i]\rangle\), 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 \(\langle REG, s \rangle\), 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 \(\langle ABORT, s \rangle\) and returns true, only if \(O[i]\) is not equal to \(\langle PRO, s \rangle\), 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 AbortableProArrayₖ

shared:

\[ A: \text{array of int}\] pairs \(\langle\perp,\perp\rangle\)

\(\text{REG, PRO, ABORT: const int}\) \(1, 2, 3\) respectively

<table>
<thead>
<tr>
<th>Operation reset()</th>
<th>Operation abort(int (i), int (seq))</th>
</tr>
</thead>
</table>
| 1 \(\text{for } i \leftarrow 0 \text{ to } k - 1 \text{ do} \)
| 2 \(\quad A[i] \leftarrow \langle 0, \perp \rangle \)
| 3 end |
| 4 \(\langle v, s \rangle \leftarrow A[i] \)
| 5 if \(v = \text{PRO} \) then return \(\text{false} \)
| 6 \(A[i] \leftarrow \langle \text{ABORT}, seq \rangle \)
| 7 return \(\text{true} \)

<table>
<thead>
<tr>
<th>Operation collect(int[] (X))</th>
<th>Operation promote()</th>
</tr>
</thead>
</table>
| 8 \(\text{for } i \leftarrow 0 \text{ to } k - 1 \text{ do} \)
| 9 \(\quad \langle v, s \rangle \leftarrow A[i] \)
| 10 if \(v = \text{ABORT} \land X[i] = \perp \) then
| 11 \(\quad A[i] \leftarrow \langle \text{REG}, X[i] \rangle \)
| 12 end |
| 13 end |
| 14 \(\text{for } i \leftarrow 0 \text{ to } k - 1 \text{ do} \)
| 15 \(\quad \langle v, s \rangle \leftarrow A[i] \)
| 16 if \(v = \text{REG} \) then
| 17 \(\quad A[i] \leftarrow \langle \text{PRO}, s \rangle \)
| 18 return \(\langle i, s \rangle \)
| 19 end |
| 20 end |
| 21 return \(\langle \perp, \perp \rangle \)

Figure 4.4: Sequential Specification of Type \(\text{AbortableProArray}_k\)
operation. Operation \texttt{promote()} takes no arguments, and changes the value of the first element (from left to right) in \( O \) with value \((\text{REG}, s)\), for some \( s \in \mathbb{Z} \), to value \((\text{PRO}, s)\), and returns \((i, s)\), where \( i \) is the index of that element. If there exists no element in \( O \) with value \((\text{REG}, s)\), for some \( s \in \mathbb{Z} \), then \( O \) is unchanged and the value \((\bot, \bot)\) is returned. In the context of our abortable lock, operation \texttt{promote()} is executed by some process, in order to “promote” some process registered in the array. Process \( i \) is said to be \textit{promoted} if a \texttt{promote()} operation returns \((i, s)\), 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 \texttt{reset()} operation.

In our abortable lock of Section 4.5, we need a concurrent wait-free linearizable implementation of type \texttt{AbortableProArray}_{\Delta}, where \( \Delta \) is the maximum number of processes that can access the object concurrently. We achieve this using the object \texttt{SFMSUnivConst(AbrortalProArray}_{\Delta})\), and therefore an operation of type \texttt{AbortableProArray}_{\Delta} executed using the \texttt{performFast()} method has step complexity \( O(1) \), while an operation executed using the \texttt{performSlow()} method has step complexity \( O(\Delta) \).
4.4 Array based Randomized Abortable Lock

4.4.1 Introduction

In this section we specify, implement, and prove properties of an object \texttt{RandALockArray}, which is an array based randomized implementation of type \texttt{TransferableAbortableLock}. Type \texttt{TransferableAbortableLock} provides methods \texttt{lock()} and \texttt{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 \(\Delta\) 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 \(\Delta\) is due to a subtlety, and we postpone its discussion to Section [4.5]) We now specify type \texttt{TransferableAbortableLock}.

Method \texttt{lock()} takes a single argument, pseudo-ID, with value in \([0, \ldots, n - 1]\). We denote a \texttt{lock()} method call with argument \(i\) as \texttt{lock\_i()}, but refer to \texttt{lock\_i()} as \texttt{lock()} whenever the context of the discussion is not concerned with the value of \(i\). Method \texttt{lock()} returns a non-⊥ value if a process captures the lock, otherwise it returns a ⊥ value to indicate a failed \texttt{lock()} call. A \texttt{lock()} call can fail only if the process executing it aborted during the call. Method \texttt{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}(\(j\)) returns \texttt{true} if and only if there exists a concurrent call to \texttt{lock()} that eventually returns \(j\). Otherwise method \texttt{release}(\(j\)) returns \texttt{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 \texttt{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 \( \text{lock}_i() \) 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 \( \text{release}_i() \) on its behalf. Such a “lock transfer” behavior is not permitted by objects of type \text{AbortableLock} (see Specification \[2.7.1\]). An algorithm that accesses an instance of an object of type \text{TransferableAbortableLock} must satisfy the following:

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

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

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

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

In the rest of the section we implement object \text{RandALockArray} of type \text{TransferableAbortableLock}, and prove all properties of the object. We first establish the properties that we desire from object \text{RandALockArray}. An execution of an algorithm that accesses an instance of an object \text{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
while true do
    <Remainder Section>
    if L.lock$_p$() = ⊥ then // Entry Section
        <Critical Section>
        j ← arbitrary integer
        L.release$_p$(j) // Exit Section
    end
end
```

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

(d) If a process’ call to release($j$) returns false, then it incurs $O(1)$ RMRs during the call, otherwise it incurs $O(n)$ 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 RCAScounter$_n$ called Ctr, an instance of type 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 init 0
  PawnSet: Object of type AbortableProArray init ∅
  apply: array [0...n - 1] of int init ⊥
  Role: array [0...n - 1] of int init ⊥
  Sync1, Sync2: int init ⊥,
  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 ⊥,
  flag, r: boolean init false,
  A: array [0...n - 1] of int init ⊥

Method lock(_)

  // If i satisfies the loop condition in line 2, 7, or 14, and i has received a signal to abort, then i calls abort(_)
  s ← getSequenceNo()
  2 await (apply[i].CAS((⊥, ⊥), (REG, s)))
  3 flag ← true
  4 repeat
    5 Role[i] ← Ctr.inc()
    6 if (Role[i] = PAWN) then
      7 await (apply[i] = (PRO, s) ∨ Ctr.Read() = 2)
      8 if (apply[i] = (PRO, s)) then
        9 | Role[i] ← PAWN_P
      end
    end
  11 until (Role[i] ∈ {KING, QUEEN, PAWN_P})
  12 if (Role[i] = QUEEN) then
    13 await (Sync1 = ⊥)
  15 end
  16 apply[i].CAS((REG, s), (PRO, s))
  17 if Role[i] = QUEEN then return Sync1 else return ∞

Figure 4.5: Implementation of Object RandALockArray
Method abort_i( )
18 if ¬flag then return ⊥
19 apply[i].CAS((REG, s), (PRO, s))
20 if Role[i] = PAWN then
21 if ¬PawnSet.abort(i, s) then
22 Role[i] ← PAWN_P
23 return ∞
24 end
25 else
26 if ¬Sync1.CAS(⊥, ∞) then
27 return Sync1
28 end
29 doCollect_i()
30 helpRelease_i()
31 end
32 apply[i].CAS((PRO, s), (⊥, ⊥))
33 return ⊥

Method release_i(int j)
34 r ← false
35 if Role[i] = KING then
36 if ¬Ctr.CAS(1, 0) then
37 r ← Sync1.CAS(⊥, j)
38 if r then doCollect_i()
39 helpRelease_i()
40 end
41 end
42 if Role[i] = QUEEN then
43 helpRelease_i()
44 end
45 if Role[i] = PAWN_P then
46 doPromote_i()
47 end
48 (dummy, s) ← apply[i]
49 apply[i].CAS((PRO, s), (⊥, ⊥))
50 return r

Method doCollect_i()
51 for k ← 0 to n − 1 do
52 (val, seq) ← apply[k]
53 if val = REG then A[k] ← seq else A[k] ← ⊥
54 end
55 PawnSet.collect(A)

Method helpRelease_i()
56 if ¬Sync2.CAS(⊥, i) then
57 j ← Sync1.Read()
58 Sync1.CAS(j, ⊥)
59 j ← Sync2.Read()
60 Sync2.CAS(j, ⊥)
61 PawnSet.remove(j)
62 doPromote_i()
63 end

Method doPromote_i()
64 PawnSet.remove(i)
65 (j, seq) ← PawnSet.promote()
66 if j = ⊥ then
67 PawnSet.reset()
68 Ctr.CAS(2, 0)
69 else
70 apply[j].CAS((REG, seq), (PRO, seq))
71 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 AbortableProArray<\textsuperscript{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 $O(\Delta)$ registers just to find a busy-waiting process
at a node, where $\Delta$ 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 $O(\Delta)$ 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 $O(1)$ 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 RCAScounter, $\text{Ctr}$, to help decide if there are any busy-waiting processes at the node. Initially, $\text{Ctr}$ is 0, and processes attempt to increase $\text{Ctr}$ using the $\text{Ctr}.\text{inc}()$ operation after having registered in the apply array. Process $p$ attempts to release the lock of a node by first executing a $\text{Ctr}.\text{CAS}(1, 0)$ operation. If the operation fails then some process $q$ must have further increased $\text{Ctr}$ 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 $O(\Delta)$ additional steps apart from the $O(\Delta)$ 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 $\text{king}_L$, $\text{queen}_L$ and $\text{ppawn}_L$, respectively. As mentioned earlier, a
RCAScounter\textsubscript{2} instance Ctr is used by processes to determine their role at lock L. Recall that RCAScounter\textsubscript{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 \textit{king}_L. The process that increases Ctr from 1 to 2 assumes the role of the queen process of lock L, and thus becomes \textit{queen}_L. By specification of object RCAScounter\textsubscript{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 \textit{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, \textit{king}_L, becomes the first owner of lock L during the current Ctr-cycle, and can proceed to enter its Critical Section, and thus it does not busy-wait during \texttt{lock()}. The queen process, \textit{queen}_L, must wait for \textit{king}_L to finish its Critical Section, and only then can it enter its Critical Section. Then \textit{queen}_L spins on the shared register \texttt{Sync1}, waiting for \textit{king}_L to write some integer value into \texttt{Sync1}. Process \textit{king}_L attempts to write an integer \(j\) into \texttt{Sync1} only during its call to \texttt{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 \(\text{king}_L\) during a call to `release()`, or by \(\text{queen}_L\) 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 \(\text{king}_L\), \(\text{queen}_L\) or \(\text{ppawn}_L\).

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

Aborting an attempt at lock \(L\) by \(\text{queen}_L\). 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, `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 `king_L` and `queen_L` are the first two processes at `L`, `king_L` will eventually try to handover `L` to `queen_L`. To prevent `king_L` from handing over lock `L` to `queen_L`, `queen_L` attempts to swap a special value `∞` into `Sync1` in one atomic step. If `queen_L` fails then this implies that `king_L` has already handed over `L` to `queen_L`, and thus `queen_L` returns from its call to `abort()` with the value written to `Sync1` by `king_L`, and becomes the owner of `L`. If `queen_L` succeeds then `queen_L` is said to have successfully aborted, and thus `king_L` will eventually fail to hand over lock `L`. Since `queen_L` has aborted, `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, `queen_L` then synchronizes with `king_L` again, to perform a promote, where one of the collected processes is promoted. After that, `queen_L` deregisters from the `apply` array by resetting its slot to the initial value `⟨⊥, ⊥⟩`.

**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 \(\bot, \bot\), and returns \(\bot\) from its call to \texttt{abort()}. 

**Releasing lock L.** Releasing lock \(L\) can be thought of as a group effort between the \(\text{king}_L\), \(\text{queen}_L\) (if present at all), and the promoted pawns (if present at all). To completely release lock \(L\), the owner of \(L\) needs to reset \(\text{Ctr}\) back to 0 for the next \(\text{Ctr}\)-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 \(\text{king}_L\), \(\text{queen}_L\) and the promoted processes. To release lock \(L\), the owner of \(L\) executes a call to \texttt{release(j)}, for some integer \(j\).

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

If \(\text{king}_L\)'s attempt to decrease \(\text{Ctr}\) from 1 to 0 fails, then \(\text{king}_L\) knows that there exists a \(\text{queen}_L\) process that increased \(\text{Ctr}\) from 1 to 2. Since \(\text{queen}_L\) is allowed to abort, releasing lock \(L\) is not as straightforward as raising a flag to be read by \(\text{queen}_L\). Therefore, \(\text{king}_L\) attempts to synchronize with \(\text{queen}_L\) by swapping the integer \(j\) into the object Sync1 using a \texttt{Sync1.CAS(\bot, j)} operation. Recall that \(\text{queen}_L\) also attempts to swap a special value \(\infty\) into object Sync1 using a \texttt{Sync1.CAS(\bot, j)} operation, in order to abort its attempt. Clearly only one of them can succeed. If \(\text{king}_L\) succeeds, then \(\text{king}_L\) is said to have successfully handed over lock \(L\) to \(\text{queen}_L\). If \(\text{king}_L\) fails, then \(\text{king}_L\) knows that \(\text{queen}_L\) has aborted and thus \(\text{king}_L\) then tries to hand over its lock to one of the waiting pawn processes. The procedure to hand over lock \(L\) to 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 \(\text{Ctr}\)-cycle, and thus we let
the process (among $\text{king}_L$ or $\text{queen}_L$) that successfully swaps a value into $\text{Sync}_1$, execute the collect action.

If $\text{king}_L$ successfully handed over $L$ to $\text{queen}_L$, it collects the waiting pawn processes, so that eventually when $\text{queen}_L$ is ready to release lock $L$, $\text{queen}_L$ can simply execute a promote action. Since there is no guarantee that $\text{king}_L$ will finish collecting before $\text{queen}_L$ desires to execute a promote action, the processes synchronize among themselves again, to execute the first promote action of the current $\text{Ctr}$-cycle. They both attempt to swap their pseudo-IDs into an empty CAS object $\text{Sync}_2$, 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 $\text{Sync}_1$ and $\text{Sync}_2$ 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 $\text{Ctr}$-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 $\text{PawnSet}$ object, and then resets $\text{Ctr}$ from 2 to 0 in one atomic step, thus releasing lock $L$, and resetting the $\text{Ctr}$-cycle.

**The release of lock $L$ by $\text{ppawn}_L$.** If a process was promoted by $\text{king}_L$ or $\text{queen}_L$ 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 $\text{Ctr}$-cycle. Since a collect for this $\text{Ctr}$-cycle has already been executed, process $\text{ppawn}_L$ 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 $\text{PawnSet}$ object, and then resets $\text{Ctr}$ from 2 to 0 in one atomic step, thus releasing lock $L$, and resetting the $\text{Ctr}$-cycle.
All owner processes also *deregister* themselves from lock L, by resetting their slot in the apply array to the initial value $\langle \bot, \bot \rangle$. 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 Sync1 during their release($j$) call. Note that only 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 $\langle \text{REG}, s \rangle$ 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 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 $\text{Role}[i]$, 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 $\text{release}_{i}(\cdot)$ on behalf of $p$, $q$ needs to determine $p$'s role at lock $L$, which is possible by reading $\text{Role}[i]$.

If the $\text{Ctr.inc()}$ 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 $\text{Ctr}$ (from the semantics of a $\text{RCAScounter}_2$ object), and it becomes $\text{king}_L$ or $\text{queen}_L$, respectively, and breaks out of the role-loop in line 12.

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

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

If the value returned in line 5 is 2, then $p$ does not become $\text{king}_L$ or $\text{queen}_L$, and thus $p$ assumes the role of a pawn. Process $p$ then waits for a notification of its own promotion, or, for the $\text{Ctr}$ value to decrease from 2, by spinning on $\text{apply}[i]$ and $\text{Ctr}$ 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 \texttt{apply}[i] was changed to \langle \texttt{PRO}, s \rangle. A process is promoted only by a \texttt{king}_L, \texttt{queen}_L or a \texttt{ppawn}_L during their \texttt{release()} call. If \( p \) finds that it was not promoted, then \( p \) is said to have been \textit{missed} during a \texttt{Ctr}-cycle, and thus \( p \) repeats the role-loop. We later show that a process gets missed during at most one \texttt{Ctr}-cycle.

If \( p \) was promoted, then it writes a constant value \texttt{PAWN}_P = 3 into \texttt{Role}[i] in line 9 and becomes \texttt{ppawn}_L. Since \( p \) has been promoted, \( p \) knows that both \texttt{king}_L and \texttt{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 \texttt{lock()} call by returning the special value \( \infty \) (a non-\( \perp \) value indicating a successful \texttt{lock()} call), since \( p \) is \texttt{ppawn}_L.

\textbf{The \texttt{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() has been completed.

In line 34 \( p \) initializes the local variable \( r \) to the boolean value \texttt{false}. Local variable \( r \) is returned later in line 50 to indicate whether the integer \( j \) was successfully written to \texttt{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 \texttt{apply}[i]. At the end of the method call a boolean is returned in line 50 indicating whether the integer \( j \) was written to \texttt{Sync1}.

If \( p \) determines that it is \texttt{king}_L, then it attempts to decrease \texttt{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 \texttt{Ctr} to 2 during its \texttt{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 \texttt{queen}_L to perform the collect action. Recall that \texttt{CAS} object \texttt{Sync1} is used by \texttt{king}_L and
queen_L to determine which process performs a collect. In line 37, p attempts to swap integer j into Sync1 by executing a Sync1.CAS(⊥, 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 doCollect_i() in line 38. If p is unsuccessful then it knows that queen_L will perform a collect. In line 39, p calls the helpRelease_i() method to synchronize the release of lock L with queen_L. We describe the method helpRelease() shortly.

If p determines that it is queen_L, then it calls the helpRelease_i() method call in line 43 to synchronize the release of lock L with 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 doCollect_i() 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 ⊥ (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 AbortableProArray_n instance PawnSet in line 55 using a PawnSet.collect(A) operation. The PawnSet.collect(A) operation ensures that if the k-th element of PawnSet has value 3 = ABORT (written during a PawnSet.abort(k, ·) operation), then the k-th element is not overwritten during the PawnSet.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 king_L calls helpRelease_i() and queen_L calls helpRelease_k(). During the course of these method calls, king_L and queen_L synchronize with each other in order to reset CAS objects Sync1 and Sync2, remove themselves from 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 kingL and queenL 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 \( \bot \). 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 \( \bot \) 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 doPromote,(). 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 \( \langle \text{REG}, \cdot \rangle \). If a process has aborted then its corresponding element in PawnSet will have the value \( \langle \text{PRO}, \cdot \rangle \).

If a successful promote() operation is executed then an element in PawnSet is changed from \( \langle \text{REG}, s \rangle \) to \( \langle \text{PRO}, s \rangle \), 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 \texttt{PawnSet} has the value \((\text{REG}, s)\), where \(s \in \mathbb{N}\), and thus the special value \((\bot, \bot)\) is returned. We then say that no process was promoted. The returned pair is stored in local variables \((j, \text{seq})\) in line 65.

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

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

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

If process \(p\) determines that it is a pawn then it attempts to remove itself from \texttt{PawnSet} by executing a \texttt{PawnSet.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 PawnSet is changed to a value \((\text{ABORT}, s)\), 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 has been promoted, and assumes the role of a promoted pawn, and in line 22 \( p \) writes \( \text{PAWN}_P \) into \( \text{Role}[i] \) and returns the special value \( \infty \) in line 23.

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

If \( p \) was unsuccessful at swapping value \( \infty \) into Sync1 then it knows the \( \text{king}_L \) is executing \( \text{release}() \), and \( \text{king}_L \) 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 Sync1 in line 27.

Process \( p \) executes line 32 only if \( p \) successfully aborted earlier in its \( \text{abort}() \) call, and thus it deregisters itself from lock \( L \) by swapping \((\bot, \bot)\) into apply\([i]\). Finally, in line 33 \( p \) returns \( \bot \) to indicate a successful abort (i.e., a failed \( \text{lock}() \) 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 \( \text{RandALockArray} \), where Conditions 4.4.1 hold. Then the following claims hold for history \( H \).

**Lemma 4.4.1.** Methods \( \text{release}_i(j) \), \( \text{abort}_i() \), \( \text{helpRelease}_i() \), \( \text{doCollect}_i() \), \( \text{doPromote}_i() \) are wait-free.
Proof. Follows from an inspection of these methods.

Claim 4.4.1. No two release$_i()$ 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 release$_i()$ concurrently for the first time at time $t$. Then from Condition 4.4.1(b)-(c), it follows that two successful calls to lock$_i()$ were executed before $t$. From condition 4.4.1(a) it follows that the two successful lock$_i()$ calls did not overlap. Consider the first successful lock$_i()$ call executed by some process $p$. Since the lock$_i()$ call returned a non-$\bot$ value, the method did not return from line 18. Then $p$ did not abort while busy-waiting in line 2 and thus apply[$i$] was set to a non-$\langle \bot, \bot \rangle$ value in line 2 during the first lock$_i()$ call. Let $t'$ be the point in time when apply[$i$] was set to a non-$\langle \bot, \bot \rangle$ value in line 2. We now show that the apply[$i$] = $\langle \bot, \bot \rangle$ in the duration between $[t', t]$.

Suppose not, i.e., some process resets apply[$i$] to $\langle \bot, \bot \rangle$ during $[t', t]$. Now, apply[$i$] is reset to a $\langle \bot, \bot \rangle$ value only in line 32 during abort$_i()$ or in line 49 during release$_i()$.

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

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

From Claim 4.4.1 and Condition 4.4.1(a) it follows that no two calls to $\text{lock}_p()$ or $\text{release}_p()$ are executed concurrently for the same $p$, where $p \in \{0, \ldots, n - 1\}$. Then we can label the process executing a $\text{lock}_p()$ or $\text{release}_p()$ 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) Role$p$ is changed by process $q$, only if $q = p$.

(b) Role$p$ is unchanged during $\text{release}_p()$.

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

(d) Role$p$ is set to value PAWN.P only when $p$ executes line 9 during $\text{lock}_p()$ or when $p$ executes line 22 during $\text{abort}_p()$.

Proof. All claims follow from an inspection of the code. \hfill \Box

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

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

(c) The $i$-th entry of PawnSet can be changed to $(\text{PRO, } s) = (2, s)$, where $s \in \mathbb{N}$, only when a process executes a PawnSet.promote$(A)$ operation in line 65.
(d) The $i$-th entry of PawnSet can be changed to $(\text{ABORT}, s) = (3, s)$, 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 $\text{AbortableProArray}_n$. \hfill $\square$

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

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

(b) $\text{apply}[p]$ is changed to value $(\text{REG}, s)$ only when process $p$ executes a successful $\text{apply}[p].\text{CAS}((\bot, \bot), (\text{REG}, s))$ operation in line 4.

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

Proof. Parts (a), (b) and (c) follow from an inspection of the code. \hfill $\square$

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 \( \text{apply}[p].\text{CAS}(\langle \bot, \bot \rangle, \langle \text{REG}, s \rangle) \) operation in line 16. Process \( p \) captures and wins lock \( L \) when it returns from \( \text{lock}_p() \) with a non-\( \bot \) value. Process \( p \) is said to promote another process \( q \) if \( p \) executes a \( \text{PawnSet}.\text{promote}() \) 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 \( \text{PawnSet}.\text{promote}() \) 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 \( \text{CAS} \) operation \( \text{L.Sync1}.\text{CAS}(\bot, j) \) in line 37, where \( q \) is the process that last increased \( \text{Ctr} \) from 1 to 2. Process \( p \) is said to have released lock \( L \) by executing a successful \( \text{Ctr}.\text{CAS}(1, 0) \) operation in line 36 or by executing a successful \( \text{Ctr}.\text{CAS}(2, 0) \) operation in line 68. Process \( p \) either hands over, promotes a process, or releases lock \( L \) during a call to \( \text{L.release}_p(j) \) 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 \( \text{apply}[p].\text{CAS}(\langle \text{PRO}, s \rangle, \langle \bot, \bot \rangle) \) operation in line 32 or 49. A process \( p \) is said to be not registered in \( \text{PawnSet} \) if the \( p \)-th entry of \( \text{PawnSet} \) is not value \( \langle \text{REG}, s \rangle \), 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.
Figure 4.7: Legend for Figures 4.8 to 4.16

- **\( \ell \)**: Atomic shared memory operation executed in line \( \ell \).
- **\( m \)**: Method call \( m \) executed in line \( \ell \).
- Operation in line \( \ell_2 \) is executed only after the operation in line \( \ell_1 \).
- Predicate \( P \) is evaluated in line \( \ell \).
  - The next line executed is \( \ell_1 \) if and only if \( P \) is true.
  - The next line executed is \( \ell_2 \) if and only if \( P \) is false.
- Predicate \( P \) is evaluated in line \( \ell \).
  - The next line executed is \( \ell_1 \) if and only if \( P \) is true.
  - The next line executed is \( \ell_2 \) if and only if \( P \) is false and condition \( C \) is false.
  - The next line executed is \( \ell_3 \) if and only if \( P \) is false and condition \( C \) is true.
- Statement \( S \) holds after line \( \ell_1 \) is executed and before line \( \ell_2 \) is executed.
- Helpful statement \( S \) to improve the readability of the figure.
- Operation \( \text{op} \) executed in line \( \ell \)
Releasers of lock and Cease-release events. A process $p$ becomes a releaser of lock $L$ at time $t$ when

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

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

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

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

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

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

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

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

$\phi_p$: $p$ executes a successful $\text{Ctr}.\text{CAS}(1,0)$ (at $36_p$ during $\text{release}_p(j)$).

$\tau_p$: $p$ executes a successful $\text{Sync2}.\text{CAS}(\bot, p)$ (at $56_p$ during $\text{helpRelease}_p()$).

$\pi_p$: $p$ promotes some process $q$ (at $65_p$ during $\text{doPromote}_p()$).

$\theta_p$: $p$ executes an operation $\text{Ctr}.\text{CAS}(2,0)$ (at $68_p$ during $\text{doPromote}_p()$).

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 doCollect\(_p\)() is called only by process \(p\) in lines 29 and 38.
(b) Method helpRelease\(_p\)() is called only by process \(p\) in lines 39, 43 and 30.
(c) Method doPromote\(_p\)() is called only by process \(p\) in line 46 and in line 62 (during helpRelease\(_p\)()).
(d) If cease-release event \(\phi_p\) occurs then \(p\) is executing release\(_p\)()\((j)\).
(e) If cease-release event \(\tau_p\) occurs then \(p\) is executing helpRelease\(_p\)().
(f) If cease-release event \(\pi_p\) or \(\theta_p\) occurs then \(p\) is executing helpRelease\(_p\)() or doPromote\(_p\)().

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

Claim 4.4.7. Consider \(p\)'s \(k\)-th passage, where \(k \in \mathbb{N}\). Note that \(s = k\). If Role[\(p\)] = PAWN\(_P\) at some point in time \(t\) during \(p\)'s call to lock\(_p\)(), then some process \(q\) promoted \(p\) at \(t_q^{66}\) and \(p\) became releaser of \(L\) by condition (R2) at \(t_q^{66} < t\).

Proof. From Claim 4.4.2(d), \(p\) changes Role[\(p\)] to PAWN\(_P\) only in line 9 or line 22.
Case a - \( p \) changed \( \text{Role}[p] \) to \( \text{PAWN\_P} \) in line \( 22 \). Then \( p \)'s call to \( \text{PawnSet.abort}(p, s) \) returned \( \text{false} \) in line \( 21 \). From the semantics of the \( \text{AbortableProArray}_n \) object, it follows that the \( p \)-th entry of \( \text{PawnSet} \) was set to value \( \langle \text{PRO}, s \rangle = \langle 2, s \rangle \). From Claim 4.4.3(c), the \( p \)-th entry of \( \text{PawnSet} \) is set to value \( \langle \text{PRO}, s \rangle \) only when a \( \text{PawnSet.promote()} \) 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 \( \text{Role}[p] \) to \( \text{PAWN\_P} \) in line \( 9 \). Then \( p \) broke out of the spin loop of line \( 2 \) and thus \( \text{apply}[p] = \langle \text{REG}, s \rangle = \langle \text{PRO}, s \rangle \) at \( t_p^8 \). Since \( p \) satisfied the if-condition of line \( 8 \), it follows that \( \text{apply}[p] = \langle \text{PRO}, s \rangle \) at \( t_p^8 \). Since \( p \) does not change \( \text{apply}[p] \) to value \( \langle \text{PRO}, s \rangle \) during \( [t_p^8, t_p^8) \) it follows that some other process changed \( \text{apply}[p] \) to value \( \langle \text{PRO}, s \rangle \). Now, \( \text{apply}[p] \) is changed to value \( \langle \text{PRO}, s \rangle \) by some other process (say \( q \)) only in line \( 70 \) and thus, from the code structure, \( q \) also executed a \( \text{PawnSet.promote()} \) 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 \( t_q^{65} < t \).

\[ \square \]

Claim 4.4.8. Consider \( p \)'s \( k \)-th passage, where \( k \in \mathbb{N} \). If \( t \in \{ \left[ t_p^{18}, t_p^{35} \right), \left[ t_p^{37}, t_p^{69} \right), \left[ t_p^{53}, t_p^{82} \right), \left[ t_p^{76}, t_p^{100} \right), \left[ t_p^{18}, t_p^{35} \right), \left[ t_p^{37}, t_p^{69} \right), \left[ t_p^{53}, t_p^{82} \right), \left[ t_p^{76}, t_p^{100} \right) \}, \) then cease-release event \( \phi_p \) does not occur before time \( t \).

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

Case a - \( t \in \left[ t_p^{18}, t_p^{35} \right) \): Then \( p \) is executing \( \text{abort}_p() \) and has not yet executed a call to \( \text{release}_p() \). Since cease-release event \( \phi_p \) can occur only during \( \text{release}_p() \), cease-release event \( \phi_p \) did not occur before time \( t \).

Case b - \( t \in \left[ t_p^{37}, t_p^{69} \right) \): Then \( p \) must have failed the if-condition of line \( 36 \) and thus \( p \) executed an unsuccessful \( \text{Ctr.CAS}(1, 0) \) operation in line \( 36 \) and cease-release event \( \phi_p \) did not occur before time \( t \).
Case c - \( t \in \{ \frac{t_{43}}{p}, \frac{t_{46}}{p}, \frac{t_{43}}{p}, \frac{t_{46}}{p} \} \): From Claim 4.4.9, \( \text{Role}[p] \in \{ \text{QUEEN}, \text{PAWN}_P \} \) at \( t \). Since \( \text{Role}[p] \) is unchanged during \( \text{release}_p() \) (Claim 4.4.2(b)), it follows that \( \text{Role}[p] = \text{KING} \) at \( \frac{t_{43}}{p} \). Then \( p \) fails the if-condition of line 35 and does not execute line 36 and thus cease-release event \( \phi_p \) did not occur before time \( t \).

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

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Value of ( \text{Role}[p] )</th>
<th>Time</th>
<th>Value of ( \text{Role}[p] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{t_1}{p} )</td>
<td>( { \bot, \text{KING}, \text{QUEEN}, \text{PAWN} } )</td>
<td>( \frac{t_{43}}{p}, \frac{t_{45}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
</tr>
<tr>
<td>( \frac{t_{13}}{p} )</td>
<td>( \text{PAWN} )</td>
<td>( \frac{t_{49}}{p}, \frac{t_{50}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
</tr>
<tr>
<td>( \frac{t_{32}}{p} )</td>
<td>( \text{PAWN}_P )</td>
<td>( \frac{t_{54}}{p}, \frac{t_{55}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN} } )</td>
</tr>
<tr>
<td>( \frac{t_{33}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
<td>( \frac{t_{58}}{p}, \frac{t_{59}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN} } )</td>
</tr>
<tr>
<td>( \frac{t_{34}}{p} )</td>
<td>( \text{QUEEN} )</td>
<td>( \frac{t_{60}}{p}, \frac{t_{61}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
</tr>
<tr>
<td>( \frac{t_{34}}{p}, \frac{t_{35}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
<td>( \frac{t_{62}}{p}, \frac{t_{63}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
</tr>
<tr>
<td>( \frac{t_{35}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
<td>( \frac{t_{65}}{p}, \frac{t_{66}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN} } )</td>
</tr>
<tr>
<td>( \frac{t_{48}}{p} )</td>
<td>( { \text{QUEEN}, \text{PAWN} } )</td>
<td>( \frac{t_{68}}{p}, \frac{t_{69}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
</tr>
<tr>
<td>( \frac{t_{49}}{p} )</td>
<td>( \text{PAWN} )</td>
<td>( \frac{t_{70}}{p}, \frac{t_{71}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN} } )</td>
</tr>
<tr>
<td>( \frac{t_{50}}{p} )</td>
<td>( \text{PAWN}_P )</td>
<td>( \frac{t_{72}}{p}, \frac{t_{73}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
</tr>
<tr>
<td>( \frac{t_{51}}{p} )</td>
<td>( \text{QUEEN} )</td>
<td>( \frac{t_{74}}{p}, \frac{t_{75}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN} } )</td>
</tr>
<tr>
<td>( \frac{t_{52}}{p} )</td>
<td>( \text{PAWN} )</td>
<td>( \frac{t_{76}}{p}, \frac{t_{77}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
</tr>
<tr>
<td>( \frac{t_{53}}{p} )</td>
<td>( \text{QUEEN} )</td>
<td>( \frac{t_{78}}{p}, \frac{t_{79}}{p} )</td>
<td>( { \text{KING}, \text{QUEEN}, \text{PAWN}_P } )</td>
</tr>
</tbody>
</table>

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

(a) If process \( p \) calls \( \text{helpRelease}_p() \) or \( \text{doPromote}_p() \) during \( \text{abort}_p() \) then it does not call \( \text{release}_p(j) \).

(b) Process \( p \) calls \( \text{helpRelease}_p() \) at most once.

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

Proof of (b): From Part (a), if $\text{helpRelease}_p()$ is executed during $\text{abort}_p()$ then $\text{release}_p(j)$ is not executed. Then to prove our claim we need to show that $\text{helpRelease}_p()$ is called at most once during $\text{abort}_p()$ and $\text{release}_p(j)$, respectively. From Claim 4.4.6(b), method $\text{helpRelease}_p()$ is called by $p$ only in lines 39, 43 and 30. Since $\text{helpRelease}_p()$ is called only once during $\text{abort}_p()$ (specifically in line 30), it follows immediately that $p$ executes $\text{helpRelease}_p()$ at most once during $\text{abort}_p()$.

From Claim 4.4.9, $\text{Role}[p] \in \{\text{KING, QUEEN, PAWN}\}$ at $t^{33}_p$. Since $\text{Role}[p]$ is unchanged during $\text{release}_p()$ (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 $\text{helpRelease}_p()$ at most once during $\text{release}_p(j)$.

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

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

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

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

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

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 $\text{release}_p(j)$, or $t \in \{ (t_p^{20}, t_p^{29}), (t_p^{29}, t_p^{38}), (t_p^{38}, t_p^{45}), (t_p^{65}, t_p^{68}), (t_p^{68}, t_p^{69}), (t_p^{69}, t_p^{70}) \}$. 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 $t \in [t_p^{35}, t_p^{38}]$ then $p$ is executing $\text{doCollect}()$. From Claim 4.4.6(a), $p$ calls $\text{doCollect}()$ only in lines 29 and 38. Then if $t \in [t_p^{35}, t_p^{38}]$ then $t \in [t_p^{29}, t_p^{39}]$ or $t \in [t_p^{65}, t_p^{68}]$. Therefore, assume now $t \in [t_p^{29}, t_p^{39}]$ or $t \in [t_p^{65}, t_p^{68}]$.

Case a - $t \in \{ (t_p^{29}, t_p^{39}), (t_p^{65}, t_p^{68}) \}$: If $t \in [t_p^{29}, t_p^{39}]$ then from a code inspection, $p$ is executing $\text{abort}_p()$ and $p$ did not execute a call to $\text{doPromote}_p()$ or $\text{helpRelease}_p()$.  

before time \( t \). If \( t \in [\frac{\tau_p}{p}, \frac{\pi_p}{p}] \) then \( p \) is executing \( \text{release}_p(j) \) and then from a code inspection and Claim 4.4.10(a) it follows that \( p \) did not execute a call to \( \text{doPromote}_p() \) or \( \text{helpRelease}_p() \) before time \( t \). Then from Claims 4.4.6(a) and 4.4.6(b) it follows that events \( \tau_p, \pi_p, \) and \( \theta_p \) did not occur before time \( t \). Since \( t \in [\frac{\pi_p}{p}, \frac{\theta_p}{p}] \) or \( t \in [\frac{\tau_p}{p}, \frac{\pi_p}{p}] \), it follows from Claim 4.4.8 that cease-release event \( \phi_p \) did not occur before time \( t \).

**Case b - \( t \in \{ [\frac{\tau_p}{p}, \frac{\pi_p}{p}), [\frac{\pi_p}{p}, \frac{\theta_p}{p}] \} \):** Then \( p \) is executing \( \text{helpRelease}_p() \). Then from Claim 4.4.6(b) it follows that \( p \) is executing a call to \( \text{helpRelease}_p() \) in line 39, 43 or 30. Then from Claim 4.4.8 it follows that cease-release event \( \phi_p \) did not occur before time \( t \). From Claim 4.4.10(b), it follows that this is \( p \)'s only call to \( \text{helpRelease}_p() \).

From Claim 4.4.6(c), \( p \) calls \( \text{doPromote}_p() \) only in line 46 and in line 62 (during \( \text{helpRelease}_p() \)). Since \( p \) has not yet executed line 46 and this is the only call to \( \text{helpRelease}_p() \), \( p \) has not called \( \text{doPromote}_p() \) before time \( t \). Then from Claim 4.4.6(b) it follows that events \( \pi_p \) and \( \theta_p \) did not occur before time \( t \). By definition, cease-release event \( \tau_p \) occurs when \( p \) executes a successful \( \text{Sync2.CAS}(\bot, p) \) in line 56. If \( t = \frac{\pi_p}{p} \), then clearly cease-release event \( \tau_p \) did not occur before time \( t \). If \( t \in [\frac{\pi_p}{p}, \frac{\theta_p}{p}] \), then \( p \) satisfied the if-condition of line 56 and thus \( p \) executed an unsuccessful \( \text{Sync2.CAS}(\bot, p) \) operation in line 56, and thus cease-release event \( \tau_p \) did not occur before time \( t \).

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

Then from Claim 4.4.8 it follows that cease-release event $\phi_p$ did not occur before time $t$.

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

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

**Subcase c(ii) -** $p$ called $\text{doPromote}_p()$ in line 46. From Claim 4.4.9, $\text{Role}[p] \in \text{PAWN}_P$ at $t^{39}_p$. Since $\text{Role}[p]$ is unchanged during $\text{release}_p()$ (Claim 4.4.2(b)), it follows that $\text{Role}[p] = \text{PAWN}_P$ at $t^{39}_p$ and $t^{42}_p$. Then $p$ fails the if-conditions of lines 35 and 42 and does not execute a call to $\text{helpRelease}_p()$ before time $t$. Then from Claim 4.4.6(c) it follows that cease-release event $\tau_p$ 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 $\text{Role}[p] \in \{ \text{KING}, \text{QUEEN}, \text{PAWN}_P \}$ at time $t$. Let $t'$ be the point when $p$ is poised to execute $\text{release}_p(j)$. From the inspection of the various points in time chosen for $t$ (including $t^{39}_p$, but excluding $t'$) and the table in Claim 4.4.9 it follows that $\text{Role}[p] \in \{ \text{KING}, \text{QUEEN}, \text{PAWN}_P \}$ at time $t$ (including $t^{39}_p$, but excluding $t'$).

Clearly $\text{Role}[p]$ is unchanged during $[t', t^{44}_p]$. Then the value of $\text{Role}[p]$ at $t'$ is the same as that at $t^{44}_p$, i.e., $\text{Role}[p] \in \{ \text{KING}, \text{QUEEN}, \text{PAWN}_P \}$.

**Case a -** $\text{Role}[p] \in \{ \text{KING}, \text{QUEEN} \}$ at time $t$: From Claim 4.4.2(c), $\text{Role}[p]$ is set to $\text{KING}$ or $\text{QUEEN}$ only when $p$ executes line 5. Then $p$ changed $\text{Role}[p]$ to $\text{KING}$ or $\text{QUEEN}$ at $t^{30}_p$ and thus $p$ became a releaser of lock $L$ by condition (R1) at $t^{30}_p = t' < t$.

**Case b -** $\text{Role}[p] = \text{PAWN}_P$ at time $t$: From Claim 4.4.7 it follows that some process $q$ promoted $p$ at $t^{45}_q$ and $p$ became a releaser of $L$ by condition (R2) at $t^{45}_q = t' < t$.  

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 $\phi_p$ occurs when $p$ executes a successful $\text{Ctr.CAS}(1, 0)$ operation in line 36 cease-release event $\tau_p$ occurs when $p$ executes a successful $\text{Sync2.CAS}(\bot, p)$ in line 56 cease-release event $\pi_p$ occurs only when $p$ executes a $\text{PawnSet.promote()}$ operation that returns a non-$\langle\bot, \bot\rangle$ value in line 65 cease-release event $\theta_p$ occurs only when $p$ executes a $\text{Ctr.CAS}(2, 0)$ operation in line 68. From Claim 4.4.11(b), $p$ is a releaser of $L$ at $t_p^0$, $t_p^0$, $t_p^0$, and $t_p^0$. Hence, the claim follows. 

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

Claim 4.4.13. Only releasers of $L$ have write-access to $\text{Sync1}$, $\text{Sync2}$ and registration-access to $\text{PawnSet}$.

Proof. The following observations follow from an inspection of the code. A value can be written to $\text{Sync1}$ only in lines 26, 37 and 58. A value can be written to $\text{Sync2}$ only in lines 56 and 60. From the semantics of the $\text{AbortableProArray}_n$ object, only operations $\text{collect}()$, $\text{promote}()$, and $\text{reset}()$ can write values in
\{(a, b) | a \in \{0, \text{REG}, \text{PRO}\}, b \in \mathbb{N}\} \text{ to } \text{PawnSet}. \text{ From Claim}\ref{claim4.3.4}, \text{ the operations } \text{collect()}, \text{ promote()}, \text{ and reset()} \text{ are executed on PawnSet only in lines }\ref{line50}-\ref{line63} \text{ and }\ref{line67} \text{ respectively.}

Suppose an arbitrary process \(p\) writes a value to \text{Sync1} or \text{Sync2}, or a value in \{(a, b) | a \in \{0, 1, 2\}, b \in \mathbb{N}\} \text{ to an entry of PawnSet}. \text{ From Claim}\ref{claim4.4.11b}, \(p\) \text{ is a releaser of } \text{L} \text{ at } t_{p}^{20}, t_{p}^{22}, t_{p}^{26}, t_{p}^{27}, t_{p}^{28}, t_{p}^{30}, t_{p}^{35}, t_{p}^{37}, \text{ and } t_{p}^{38}. \text{ Hence, the claim follows.} \qed

\textbf{Claim 4.4.14.} \textit{The } \(i\)-entry \text{ of PawnSet can be changed only by process } i \text{ or a releaser of } \text{L}.\]

\textit{Proof.} The values that can be written to PawnSet are in \{(a, b) | a \in \{0, 1, 2, 3\}, b \in \mathbb{N}\}.\ A process that can write values in \{(a, b) | a \in \{0, 1, 2\}, b \in \mathbb{N}\} \text{ to any entry of PawnSet is said to have registration-access to PawnSet}. \text{ From Claim}\ref{claim4.4.13} \text{ it follows that only a releaser of } \text{L} \text{ has registration-access to PawnSet, therefore only a releaser of } \text{L} \text{ can write values in } \{(a, b) | a \in \{0, 1, 2\}, b \in \mathbb{N}\} \text{ to the } i \text{-th entry of PawnSet}. \text{ From Claim}\ref{claim4.4.3d} \text{ the value } \langle \text{ABORT}, s \rangle = \langle 3, s \rangle, \text{ where } s \in \mathbb{N}, \text{ can be written to the } i \text{-th entry of PawnSet only when a process executes a } \text{remove}(i), \text{ remove}(i) \text{ or PawnSet.} \text{abort}(i, s) \text{ operation in line }\ref{line64}\text{ or }\ref{line61} \text{ respectively. From Claim}\ref{claim4.4.11b}, \text{ it follows that a process executing lines }\ref{line64}\text{ and }\ref{line61} \text{ is a releaser of } \text{L}. \text{ Since a PawnSet.} \text{abort}(i, s) \text{ operation in line }\ref{line21} \text{ is executed only by process } i, \text{ our claim follows.} \qed

\textbf{Claim 4.4.15.} \textit{Sync2 is changed to a non-\(\perp\) value only by a releaser of } \text{L} \text{ (say } r) \text{ in line }\ref{line56} \text{ which triggers the cease-release event } \tau_{r}.\]

\textit{Proof.} By definition, cease-release event \(\tau_{p}\) occurs when \(p\) executes a successful Sync2.\text{CAS}(\perp, p) \text{ in line }\ref{line56}. \text{ From a code inspection, Sync2 is changed to a non-\(\perp\) value only when some process (say } r) \text{ executes a successful Sync2.} \text{CAS}(\perp, r) \text{ operation in line }\ref{line56}. \text{ From Claim}\ref{claim4.4.13} \text{ it follows that Sync2 is changed only by a releaser of } \text{L}. \text{ Then } r \text{ is a}
releaser of L when it changes Sync2 to a non-$\bot$ value in line 56 and doing so triggers the cease-release event $\tau_r$.

Claim 4.4.16. A PawnSet.promote() operation is executed only by a releaser of L (say $r$), and if the value returned is non-{$\bot$, $\bot$} the cease-release event $\pi_r$ is triggered.

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

Claim 4.4.17. During an execution of doPromote$_p()$ exactly one of the events $\pi_p$ and $\theta_p$ occurs.

Proof. By definition, cease-release event $\pi_p$ occurs when $p$ executes a PawnSet.promote() operation in line 65 that returns a non-{$\bot$, $\bot$} value, and cease-release event $\theta_p$ occurs when $p$ executes a Ctr.CAS(2,0) operation in line 68 during doPromote$_p()$.

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

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

Claim 4.4.18. During an execution of helpRelease$_p()$ exactly one of the events $\tau_p$, $\pi_p$ and $\theta_p$ occurs.
Proof. By Claim 4.4.5, events $\pi_p$ and $\theta_p$ can only occur during $p$’s call to $\text{doPromote}_p()$, and cease-release event $\tau_p$ occurs when $p$ executes a successful $\text{Sync2.CAS}(\perp, p)$ operation in line 56.

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

Case b - $p$ executes an unsuccessful $\text{Sync2.CAS}(\perp, p)$ operation in line 56 and thus cease-release event $\tau_p$ does not occur. Then $p$ satisfies the if-condition of line 56 and calls $\text{doPromote}_p()$ in line 62. From Claim 4.4.17 exactly one of the events $\pi_p$ and $\theta_p$ occurs during $p$’s call to $\text{doPromote}_p()$. \qed

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

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

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

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

$\text{Ctr}$-Cycle Interval $T$. Let $T = [t_s, t_e)$ be a time interval where $t_s$ is a point when $\text{Ctr}$ is 0 and $t_e$ is the next point in time when $\text{Ctr}$ is decreased to 0. For $i \in \{0, 1, 2\}$
let \( I_i = \{ t \in T | \text{Ctr} = i \text{ at } t \} \) and let time \( I_i^- = \min(I_i) \) and time \( I_i^+ = \max(I_i) \).

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 \( I_2 = \emptyset \) if and only if \( \text{Ctr} \) is never increased to 2 during \( T \). Moreover, \( t_s = I_0^- \) and \( I_0 \) is immediately followed by \( I_1 \) (i.e., \( \min(I_1) = \max(I_0) + 1 \)). If \( I_2 = \emptyset \) then \( I_2 \) follows immediately after \( I_1 \). The \( \text{Ctr} \)-cycle interval \( T \) ends either at time \( I_1^+ \) if \( I_2 = \emptyset \), or at time \( I_2^+ \) if \( I_2 = \emptyset \).

Then it also follows that exactly one process changes \( \text{Ctr} \) from 0 to 1 during \( T \), and it does so at time \( I_1^- \). Let \( K \) be the process that increases \( \text{Ctr} \) to 1 at time \( I_1^- \). And if \( I_2 = \emptyset \) then exactly one process changes \( \text{Ctr} \) from 1 to 2 during \( T \), and it does so at time \( I_2^- \). If \( I_2 = \emptyset \) let \( Q \) be the process that increases \( \text{Ctr} \) 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 \( R(I_0^-) = \emptyset \) and at \( I_0^- \), \( \text{Sync1} = \text{Sync2} = \bot \) and \( \text{PawnSet} \) is candidate-empty, then the following holds:

(a) \( \forall t \in I_0^- : R(t) = \emptyset \) and throughout \( I_0 \), \( \text{Sync1} = \text{Sync2} = \bot \) and \( \text{PawnSet} \) is candidate-empty.

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

(c) \( K \) executes lines of code of \( \text{lock}_K() \) starting with line 2 as depicted in Figure 4.8. 

(A legend for the figure is given in Figure 4.7.)

\[
\text{Apply}[K].\text{CAS}((\bot, 1), (\text{REG}, s))
\]

\[
\text{false \rightarrow 5 \rightarrow 6 \rightarrow 12 \rightarrow 13 \rightarrow 16 \rightarrow 17}
\]

\[
\text{Role}[K] = \text{KING} = 0
\]

\[
K \text{ returns } \infty
\]

Figure 4.8: \( K \)'s call to \( \text{lock}_K() \)

(d) \( K \)'s call to \( \text{lock}_K() \) returns \( \infty \) and \( \text{Role}[K] = \text{KING} \) throughout \([T, T]_K\).
(e) $K$ executes a $\text{Ctr.CAS}(1,0)$ operation in line 36 during $T$, and $K$ does not change $\text{Sync1}, \text{Sync2}$ or $\text{PawnSet}$ throughout $[I_1^{-}, I_2^{+}]$.

(f) $\forall t \in I_1: R(t) = \{K\}$.

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

Proof. Proof of (a): Consider the claim $R(t) = \emptyset$ where $t \in I_0$. Since $R(I_0^-) = \emptyset$ 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 $\text{Ctr}$ to 1 or 2 (condition (R1)) at time $t'$, since $\text{_ctr} = 0$ 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 $\text{Sync1}, \text{Sync2}$ and $\text{PawnSet}$ 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 $t \in I_0$, $R(t) = \emptyset$, it follows that the variables are unchanged throughout $I_0$.

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

Proof of (c) and (d): Since $K$ is the process that increased $\text{Ctr}$ from 0 to 1 at time $I_1^-$, and since $K$ can increase $\text{Ctr}$ only by executing a $\text{Ctr.inc()}$ operation in line 5 (by Claim 4.4.19) $K$ set $\text{Role[K]} = 0 = \text{KING}$ at $I_0^-$. Then from the code structure, $K$
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 $K$ proceeds to execute lines 16-17 and returns value $\infty$ in line 17. Since $K$ does not change $\text{Role} [K]$ during $[K_1^4, K_1^7]$, $\text{Role} [K] = \text{KING}$ throughout $[K_1^4, K_1^7]$.

**Proof of (e):** Since $K$ is the process that increased $\text{Ctr}$ from 0 to 1 at time $I_1^-$, and since $K$ can increase $\text{Ctr}$ only by executing a $\text{Ctr}.\text{inc}()$ operation in line 5 (by Claim 4.4.19) $K$ set $\text{Role} [K] = 0 = \text{KING}$ at $t_K^4$. From Part (d), $K$ returns from lock$_K()$ with value $\infty$ in line 17 and thus $K$ consequently calls release$_K (j)$ (follows from conditions (b) and (c)). Note that $K$ has not executed any operations on Sync1, Sync2 and PawnSet in the process. Then $\text{Role} [K] = \text{KING}$ at $t_K^{35}$ and thus $p$ satisfies the if-condition of line 35 and executes the Ctr.CAS(1, 0) operation in line 36 during $T$ without having executed any operations on Sync1, Sync2 and PawnSet in the process. Thus $K$ did not change Sync1, Sync2 or PawnSet during $[I_1^-, t_K^{36}]$.

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

**Case a -** Process $K$ ceases to be a releaser of $L$ at $t'$: By definition, cease-release event $\phi_K$ occurs when $K$ executes a successful Ctr.CAS(1, 0) operation in line 36. From Part (e), $K$ executes a Ctr.CAS(1, 0) operation in line 36. If $K$ executes a successful Ctr.CAS(1, 0) operation in line 36 then, by definition, cease-release event $\phi_K$ occurs and by Claim 4.4.12 $K$ ceases to be the releaser of $L$. Thus, $t' = t_K^{36}$ and $\text{Ctr}$ changes to value of 0 at $t'$. But since $t' \in I_1$ and $\text{Ctr} = 1$ throughout $I_1$, we have a contradiction. If $K$ executes an unsuccessful Ctr.CAS(1, 0) operation in line 36 then $\text{Ctr} = 1$ at $t_K^{36}$. Since $p$ did not cease to a releaser at $t_K^{36}$, $t_K^{36} < t'$. Since $I_1^- = t_K^4 < t_K^{36} < t' < I_1^+$ and $\text{Ctr} = 1$ throughout $I_1$, $\text{Ctr} = 1$ at $t_K^{36}$, 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 $\pi_r$ occurs at $t'$, and thus from Claim 4.4.12 it follows that $r$ is a releaser of $L$ immediately before $t'$. Since $K$ is the only releaser immediately before $t'$, $r = K$. Then cease-release event $\pi_K$ occurred at $t'$ and $K$ 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 $\text{Sync} = \text{Sync}2 = \bot$ and $\text{PawnSet}$ is candidate-empty holds by Part [b]. Suppose some process $p$ changes $\text{Sync}2$ or $\text{Sync}1$ or $\text{PawnSet}$ 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 $t \in I_1$, $R(t) = \{K\}$ (Part [h]), it follows that $p = K$. From Part [e], $K$ does not change any of the variables before the point when it executes a $\text{Ctr}.\text{CAS}(1, 0)$ operation in line 36 i.e., $t_1^{36} < t'$. If $K$ executes a successful $\text{Ctr}.\text{CAS}(1, 0)$ operation in line 36 then the interval $I_1$ ends and clearly $t' \notin I_1$, hence a contradiction. If $K$ executes an unsuccessful $\text{Ctr}.\text{CAS}(1, 0)$ operation in line 36 then $\text{Ctr} = 1$ at $t_1^{36}$. Since $I_1^- = t_1^{-} < t_1^{36} < t' < I_1^+$ and $\text{Ctr} = 1$ throughout $I_1$, we have a contradiction. ∎

Claim 4.4.22. If $I_2 = \emptyset$ and $R(I_0^-) = \emptyset$ and at $I_0^-$, $\text{Sync}1 = \text{Sync}2 = \bot$ and $\text{PawnSet}$ is candidate-empty, then the following claims hold:

(a) $R(I_2^-) = \{K, Q\}$ and at time $I_2^-$, $\text{Sync}1 = \text{Sync}2 = \bot$ and $\text{PawnSet}$ is candidate-empty.

(b) $K$ and $Q$ 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 $K$ takes enough steps, $K$ executes lines of code of $\text{release}_K()$ starting with line 34 as depicted in Figure 4.9.
(e) \( K \) executes an unsuccessful \( \text{Ctr.CAS}(1,0) \) operation in line 36 and calls helpRelease\(_{K}\)() in line 39 such that \( I_{i} < k_{K} < k_{K}^{39} \).

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

(g) If \( K \) and \( Q \) take enough steps, \( Q \) executes lines of code of lock\(_{Q}\)() starting with line 3 as depicted in Figure 4.10.

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

(i) \( Q \) calls helpRelease\(_{Q}\)() either in line 30 or in line 43 after time \( I_{i} \).

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

Figure 4.10: \( Q \)'s call to lock\(_{Q}\)( )
Figure 4.11: Q’s call to \texttt{release}_Q() \\

Proof. Proof of (a) and (b): Since Q is the process that increases Ctr from 1 to 2 at time $I_2^-$, and since Q can increase Ctr only by executing a Ctr.inc() operation in line 5 (by Claim 4.4.19), Q becomes a releaser of lock L by condition (R1) at $I_2^- = I_3$. Since for all $t \in I_1$, $R(t) = \{K\}$ (Claim 4.4.21[f]), it follows that $R(I_2^-) = \{K, Q\}$. By claim 4.4.21[g], throughout $I_1$, Sync1 = Sync2 = ⊥ and PawnSet is candidate-empty, and since the only operation executed at time $I_2^-$ is Ctr.inc(), it follows that at time $I_2^-$, Sync1 = Sync2 = ⊥ and PawnSet is candidate-empty. Hence Part (b) holds. Clearly K and 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 PawnSet.promote() operation is executed in line 55 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], K executes the Ctr.CAS(1, 0) operation in line 36 during $T$. If K’s Ctr.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 K’s Ctr.CAS(1, 0) operation is unsuccessful.

Since K executes an unsuccessful Ctr.CAS(1, 0) operation in line 36, K satisfies the if-condition of line 36, executes lines 37-38 and calls helpRelease_k() in line 39 and then
executes lines [49, 50].

Since \( K \) executes an unsuccessful \( \text{Ctr.CAS}(1, 0) \) operation in line [36], it follows that \( \text{Ctr} \) was changed from 1 to 2 at time \( I_2^- \) (by definition), and thus \( I_2^- < t_K^{39} \). Since \( t_K^{39} < t_K^{39} \), it follows that \( I_2^- < t_K^{39} = t_K^{29} \).

**Proof of (f), (g) and (h):** Since \( Q \) is the process that increases \( \text{Ctr} \) from 1 to 2 at time \( I_2^- \), and since \( Q \) can increase \( \text{Ctr} \) only by executing a \( \text{Ctr.inc()} \) operation in line [5] (by Claim [4.4.19]) \( Q \) set \( \text{Role}[K] = 1 = \text{QUEEN} \) at \( t_K^3 = I_2^- \). Then from the code structure, \( Q \) 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 \( Q \) does not finish \( \text{lock}_Q() \) only if it spins indefinitely in line [14] and does not receive a signal to abort.

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

**Case a -** \( K \) resets \( \text{Sync1} \) in line [58] before \( Q \) reads \( \text{Sync1} \) again: For \( K \) to reset \( \text{Sync1} \) in line [58], \( K \) must satisfy the if-condition of line [56] and thus \( K \) must execute an unsuccessful \( \text{Sync2.CAS}(\bot, \text{K}) \) operation in line [56]. Since \( \text{Sync2} = \bot \) at time \( I_2^- \) (Part (a)), and only a releaser can change \( \text{Sync2} \) (Claim [4.4.13]), and \( Q \) is busy-waiting in line [14], it follows that \( \text{Sync2} = \bot \) at \( t_K^{39} \). Thus \( K \)'s \( \text{Sync2.CAS}(\bot, \text{K}) \) operation in line [56] is successful and we get a contradiction.

**Case b -** some other process becomes a releaser and resets \( \text{Sync1} \) before \( Q \) reads
Sync1 again: From Part (i) 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 \(K\) is the only other releaser of \(L\) apart from \(Q\), it follows that \(K\) promotes some process before \(Q\) reads Sync1 again. As argued in Case a, \(K\) executes a successful \(\text{Sync2.CAS}(\bot, K)\) operation in line 56. Then from the code structure, \(K\) does not call \(\text{doPromote}_K()\) in line 62 and thus \(K\) does not promote any process. Hence, we have a contradiction.

Proof of (i): Since \(Q\) is the process that increases \(\text{Ctr}\) from 1 to 2 at time \(I_2^-\), and since \(Q\) can increase \(\text{Ctr}\) only by executing a \(\text{Ctr.inc}()\) operation in line 5 (by Claim 4.4.19) \(Q\) set \(\text{Role}[K] = 1 = \text{QUEEN}\) at \(t_4^Q\). Then from the code structure, \(Q\) 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 - \(Q\) does not receive a signal to abort while busy-waiting in line 14. From Part (i), \(Q\) does not busy-wait indefinitely in line 14 and eventually breaks out. Since \(Q\) breaks out of the spin loop of line 14 it reads non-\(\bot\) from Sync1 and then from the code structure it follows that \(Q\) goes on to return that non-\(\bot\) value in line 17. Consequently \(Q\) calls \(\text{release}_Q(j)\) (follows from conditions [b] and [c]). Consider \(Q\)'s call to \(\text{release}_Q(j)\). Since \(Q\) last changed \(\text{Role}[Q]\) only in line 5, \(\text{Role}[Q] = \text{QUEEN}\) at \(t_4^Q\). Since \(\text{Role}[Q]\) is unchanged during \(\text{release}_Q()\) (Claim 4.4.2(b)), it follows that \(\text{Role}[Q] = \text{QUEEN}\) throughout \(\text{release}_Q()\). Then from the code structure it follows that \(Q\) executes only lines 34-35, 42-45 and 49-50. Then \(Q\) calls \(\text{helpRelease}_Q()\) only in line 43 and since \(I_2^- = t_4^Q < t_\text{Sync1}^Q\), our claim holds.

Case b - \(Q\) receives a signal to abort while busy-waiting in line 14. Then \(Q\) calls \(\text{abort}_Q()\), and from the code structure \(Q\) executes lines 18-20 and then line 26. If \(Q\) fails the \(\text{Sync1.CAS}(\bot, \infty)\) operation of line 26 then \(\text{Sync1} = \bot\) at \(t_\text{Sync1}^Q\). From Claim 4.4.13 only a releasers of \(L\) can change Sync1 to a non-\(\bot\) value, and since \(K\) and \(Q\) are the
only releasers of L, it follows that K changed Sync1 to a non-⊥ value. Then Q satisfies
the if-condition of line 26 and returns the non-⊥ value written by K to Sync1 in line 27.
Consequently Q calls release\(_Q\)(j) (follows from conditions b and c), and as argued in
Case a, Q executes only lines 34-35, 42-45 and 49-50, and Q calls helpRelease\(_Q\)() only
in line 43. Since \(I^2_Q = I^3_Q < I^4_Q\), our claim holds.

If Q’s Sync1.CAS(⊥, ∞) operation is successful, then Q goes on to call doCollect\(_Q\)() in
line 29 calls helpRelease\(_Q\)() in line 30, then executes lines 32-33, and finally returns
⊥ in line 33. Since \(I^2_Q = I^3_Q < I^4_Q\), our claim holds. \(\square\)

Define \(\lambda\) to be the first point in time when Sync2 is changed to a non-⊥ value, and if
Sync2 is never changed to non-⊥ then \(\lambda = \infty\). Define \(\gamma\) 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 K and Q execute
helpRelease\(_K\)() and helpRelease\(_Q\)(), respectively, after time \(I^2\). Let \(A \in \{K, Q\}\) be
the first process among them to execute line 56, and let \(B \in \{K, Q\} - \{A\}\) be the other
process, i.e., \(I^5_A < I^5_B\).

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

(a) \(I^2 < \lambda = I^6_A\) and for all \(t \in [I^2, \lambda)\), \(R(t) = \{K, Q\}\) and Sync2 = ⊥ throughout
\([I^2, \lambda)\), and cease-release event \(\tau_A\) occurs at \(\lambda\).

(b) If K and Q take enough steps, then A executes lines of code of helpRelease\(_A\)()
starting with line 56 as depicted in Figure 4.12.

\[\text{Sync2.CAS}(⊥, A) \text{ succeeds}\]

\[56\]

\[\bullet\]

**Figure 4.12:** A’s call to helpRelease\(_A\)()
(c) If $\mathcal{K}$ and $\mathcal{Q}$ take enough steps, then $\mathcal{B}$ executes lines of code of $\text{helpRelease}_B()$ and $\text{doPromote}_B()$ as depicted in Figures 4.13 and 4.14, respectively.

\[ \text{Sync2.CAS}(1, B) \text{ fails} \]

Figure 4.13: $\mathcal{B}$’s call to $\text{helpRelease}_B()$

\[ \langle j, \text{seg} \rangle = \text{PawnSet.promote()} \]

\[ j = 1 \]

\[ \text{true} \]

\[ \text{false} \]

Figure 4.14: $\mathcal{B}$’s call to $\text{doPromote}_B()$

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

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

(f) At time $\gamma$, Sync1 = Sync2 = $\perp$.

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

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

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

(j) If the $\text{PawnSet.promote()}$ operation at time $\gamma$ returns value $\langle \perp, \perp \rangle$ then $\mathcal{B}$’s cease-release event $\theta_B$ occurs at $t' = t_{68}^{\mathcal{B}} \geq \gamma$, 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 $\text{doCollect}()$, specifically during $[I_2^-, \gamma]$. 

Proof. Proof of (a): We first show that for all $t \in [I_2^-, t_{K_\mathcal{A}}^{\mathcal{K}}]$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$ and then show that $\lambda = t_{K_\mathcal{A}}^{\mathcal{K}}$. From Claims 4.4.22(c) $\mathcal{K}$ calls $\text{helpRelease}_\mathcal{K}()$ in line 30 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 $[I_1, t_{K_\mathcal{A}}^{\mathcal{K}}]$ $\mathcal{K}$ does not execute a call to $\text{helpRelease}_\mathcal{K}()$ or $\text{doPromote}_\mathcal{K}()$. Also from an inspection, $\mathcal{K}$’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 $\text{helpRelease}_\mathcal{K}()$ or $\text{doPromote}_\mathcal{K}()$ (Claims 4.4.5(e) and 4.4.5(f)), it follows that $\mathcal{K}$ is a releaser of $L$ throughout $[I_1, t_{K_\mathcal{A}}^{\mathcal{K}}]$.

From Claim 4.4.22(b), $\mathcal{Q}$ calls $\text{helpRelease}_\mathcal{Q}()$, 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 $[I_2, t_{K_\mathcal{A}}^{\mathcal{Q}}]$ $\mathcal{Q}$ does not execute a call to $\text{helpRelease}_\mathcal{Q}()$ or $\text{doPromote}_\mathcal{Q}()$. Also from an inspection, $\mathcal{Q}$ does not execute a $\text{Ctr.CAS}(1, 0)$ operation in line 30 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 $\text{helpRelease}_\mathcal{Q}()$ or $\text{doPromote}_\mathcal{Q}()$ (Claims 4.4.5(e) and 4.4.5(f)), it follows that $\mathcal{Q}$ is a releaser of $L$ throughout $[I_2^-, t_{K_\mathcal{A}}^{\mathcal{Q}}]$.

Then for all $t \in [I_2^-, t_{K_\mathcal{A}}^{\mathcal{K}}]$, $\{\mathcal{K}, \mathcal{Q}\} \subseteq R(t)$ since $I_1^- < I_2^-$ and $t_{K_\mathcal{A}}^{\mathcal{K}} = t_{K_\mathcal{A}}^{\mathcal{Q}} = \min(t_{K_\mathcal{A}}^{\mathcal{K}}, t_{K_\mathcal{A}}^{\mathcal{Q}})$. 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 $t \in [I_2^-, t_{K_\mathcal{A}}^{\mathcal{K}}]$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$, we need to show that no process is promoted by $\mathcal{K}$ or $\mathcal{Q}$ during $[I_2^-, t_{K_\mathcal{A}}^{\mathcal{K}}]$. If a process was promoted by $\mathcal{K}$ or $\mathcal{Q}$ during $[I_2^-, t_{K_\mathcal{A}}^{\mathcal{K}}]$ then by definition cease-release events $\pi_\mathcal{K}$ or $\pi_\mathcal{Q}$ would have occurred during $[I_-^2, t_{K_\mathcal{A}}^{\mathcal{K}}]$, but as shown above this does not happen.

From Claim 4.4.22(a), $\text{Sync2} = \bot$ at time $I_2^-$. From a code inspection, $\text{Sync2}$ is changed to a non-$\bot$ value only in line 56 (during $\text{helpRelease}()$), moreover only by a releaser of $L$ (from Claim 4.4.13). Since for all $t \in [I_2^-, t_{K_\mathcal{A}}^{\mathcal{K}}]$, $R(t) = \{\mathcal{K}, \mathcal{Q}\}$ and
\( \lambda_A \) = \min(\lambda_A^-, \lambda_A^+) \), it follows then that \( \text{Sync2} = \bot \) throughout \([I_2^-, \lambda_A^-]\) and \( \mathcal{A} \) executes a successful \( \text{Sync2.CAS}(\bot, \mathcal{A}) \) operation in line 56. Thus \( \mathcal{A} \)'s cease-release event \( \tau_A \) occurs at \( \lambda = \lambda_A^- \).

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

**Proof of (b):** From Part (a), \( \mathcal{A} \)'s cease-release event \( \tau_A \) occurs at \( \lambda = \lambda_A^- \), and thus \( \mathcal{A} \)'s \( \text{Sync2.CAS}(\bot, \mathcal{A}) \) 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 \( \text{helpRelease}_A() \). Thus, Figure 4.12 follows.

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

Now consider \( \mathcal{B} \)'s \( \text{helpRelease}_B() \) call. Since \( \lambda = \lambda_A^- < \lambda_B^- \) and \( \text{Sync2} = \bot \) at \( \lambda \) and \( \mathcal{B} \) has exclusive write-access to \( \text{Sync2} \) throughout \([\lambda, t]\), \( \mathcal{B} \) fails the \( \text{Sync2.CAS}(\bot, \mathcal{B}) \) operation at \( \lambda_B^- \), and thus satisfies the if-condition on line 56. It then executes lines 57 - 62 and calls \( \text{doPromote}_B() \) in line 62. Then Figures 4.13 and 4.14 and Part (c) follows immediately.
We now show that $\gamma = t_B^5 \leq t$. Since $\lambda = t_A^5$ and $t_A^5 < t_B^5 < t_B^6$, it would follow that $\lambda < \gamma$, and hence we would have proved Part (d). And since $B$ is the only releaser of $L$ throughout $[\lambda, t)$, we would have proved Part (e) as well, i.e., $B$ is the only releaser throughout $[\lambda, \gamma)$.

During doPromote$_B()$, $B$ executes a PawnSet.promote() operation in line $65$. Since $K$ and $Q$ are the first two releasers of $L$ during $T$ (Claim 4.4.22(b)), and only a releaser executes a PawnSet.promote() operation (Claim 4.4.16), and $A$ ceased to be a releaser at $t_A^5 < t_B^5$, it follows that $B$’s PawnSet.promote() operation in line $65$ is the first PawnSet.promote() operation, and thus $\gamma = t_B^5$. Since none of $B$’s cease-release events occur during $[t_A^5, t_B^5]$, $t \geq t_B^5$.

During $[t_B^5, t_B^6]$, $B$ resets Sync1 and Sync2 in lines $58$ and $60$, respectively, and since $B$ has exclusive write-access to Sync1 and Sync2 throughout $[t_B^5, t_B^6]$, at time $\gamma = t_B^5$, Sync1 = Sync2 = $\bot$. Thus, Part (f) follows.

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

Since $B$ has exclusive write-access to Sync2 throughout $[\lambda, t_B^6]$, and Sync2 = $A$ at $\lambda > t_B^6$, $B$ reads the value $A$ from Sync2 in line $59$ and executes a PawnSet.remove($A$) operation in line $61$. Since $B$ executes PawnSet.remove($A$) and PawnSet.remove($B$) in lines $61$ and $64$ during $[\lambda, \gamma)$ and $B$ has exclusive registration-access to PawnSet during $[\lambda, \gamma)$, it follows from the semantics of the AbortableProArray $n$ object that $B$’s PawnSet.promote() operation at time $\gamma$ does not return values in $\{ \langle a, b \rangle | a \in \{A, B\} = \{K, Q\}, b \in \mathbb{N} \}$. Hence, Part (h) follows.

Case a - $B$’s PawnSet.promote() operation returns a non-$\langle \bot, \bot \rangle$ value: Then $B$’s cease-release event $\pi_B$ occurs at $t_B^6 = \gamma$ (Claim 4.4.16), and thus Part (i) holds.
Case b - \(B\)'s \texttt{PawnSet.promote()} operation in line 65 returns \((\perp, \perp)\). Then \(B\) did not find any process to promote, and thus cease-release event \(\pi_B\) did not occur. From the code structure \(B\) goes on to execute a \texttt{Ctr.CAS}(2, 0) operation in line 68. Since \(\text{Ctr} = 2\) throughout \(I_2\), it follows that \(B\)'s \texttt{Ctr.CAS}(2, 0) operation succeeds, and thus \(B\)'s cease-release event \(\theta_B\) occurs at \(t_B^{68}\) and the intervals \(I_2\) and \(T\) end. Therefore \(t' = t_B^{68} > t_B^{65} = \gamma\). Clearly, \(B\) does not promote any process in \([t_B^{65}, t_B^{68}] = [\gamma, t']\), and thus Part (j) holds.

Proof of (k): From an inspection of Figure 4.9 \(K\) executes a \texttt{Sync1.CAS}(\(\perp, \mathcal{K}\)) operation in line 37. Since \(I_2^- < t_K^{37} < t_K^{37} < t_K^{29} < \gamma\) (from Parts (a) and (d)), it follows that \(t_K^{37} \in [I_2^-, \gamma]\). From an inspection of Figures 4.10 and 1.10 \(Q\) may or may not execute a \texttt{Sync1.CAS}(\(\perp, \infty\)) operation in line 26. If \(Q\) executes a \texttt{Sync1.CAS}(\(\perp, \infty\)) operation in line 26, since \(I_2^- < t_Q^{26} < t_Q^{29} < \gamma\), it follows that \(t_Q^{26} \in [I_2^-, \gamma]\).

Since for all \(t \in [I_2^-, \gamma]\), \(R(t) \subseteq \{\mathcal{K}, Q\}\) (from Parts (a) and (d)), and only releasers of \(L\) have write-access to \texttt{Sync1} (Claim 4.4.13), and \texttt{Sync1} = \(\perp\) at \(I_2^-\) (Claim 4.4.22[a]), it follows that either \(\mathcal{K}\) or \(Q\) executes a successful \texttt{CAS()} operation on \texttt{Sync1}. Then from the code structure it follows that either \(\mathcal{K}\) or \(Q\) executed a call to \texttt{doCollect()} in lines 38 or 29, respectively. Since \(t_K^{29} < t_Q^{37} = t_K^{29} < \gamma\) and \(t_Q^{29} < t_Q^{37} < \gamma\), \(\mathcal{K}\) or \(Q\) executed a call to \texttt{doCollect()} during \([I_2^-, \gamma]\).

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

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

Let \( \ell \) be the number of times a promotion occurs during \( T \). For all \( i \in \{1, \ldots, \ell\} \), define \( \Omega_i \) 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 \( \Omega^-_i \).

Claim 4.4.25. If \( I_2 = \emptyset \) and \( R(I^-_0) = \emptyset \) and at time \( I^-_0 \), \( \text{Sync1} = \text{Sync2} = \perp \) and \texttt{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 \( \text{Sync1} = \text{Sync2} = \perp \) at \( \Omega^-_1 \), and no \texttt{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 through \( \Omega_i \))

(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 \( \Omega^-_{i+1} \))

(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 through \( \Omega_i \))

Proof. Proof of (a): If the \texttt{PawnSet.promote()} operation at time \( \gamma \) 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 \texttt{PawnSet.promote()} operation at time \( \gamma \) returns a non-\( \langle \perp, \perp \rangle \) value. By definition \( \gamma \) is the point when the first
PawnSet.promote() operation occurs, and $\Omega_i^-$ is the point when the first promotion occurs and $\mathcal{P}_1$ is the process promoted at $\Omega_i^-$. Then $\gamma = \Omega_i^-$, and $\mathcal{P}_1$ is the first promoted process. From Claim 4.4.23(c), $\mathcal{B}$ is the only releaser of $L$ at the point in time immediately before time $\gamma$. Then from Claim 4.4.23(f) it follows that $\mathcal{B}$ promotes $\mathcal{P}_1$ at time $\gamma = \Omega_i^-$, and $\mathcal{B}$ ceases to be a releaser of $L$ at $\gamma$, therefore $R(\gamma) = \{\mathcal{P}_1\}$. From Claim 4.4.23(f) it follows that $\text{Sync}1 = \text{Sync}2 = \bot$ at $\Omega_i^-$. 

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

Proof of (b): Since $R(\Omega_i^-) = \{\mathcal{P}_1\}$, and $\Omega_i^+$ 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 $\Omega_i^- > \Omega_i^+ = \gamma > I_2^-$, 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 $\Omega_{i+1}^- > \Omega_i^- > \Omega_i^- = \gamma > I_2^-$, 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 PawnSet.promote() in line 65 that returns $(\mathcal{P}_{i+1}, s)$, where $s \in \mathbb{N}$, and event $\pi_{\mathcal{P}_i}$ occurs at $t_{\mathcal{P}_i}$. Then $\mathcal{P}_i$ ceases to be a releaser of $L$ at $t_{\mathcal{P}_i}$ and thus $\Omega_i^+ = t_{\mathcal{P}_i}$. Since $\Omega_{i+1}^-$
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 $\Omega^-_1$, 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 $\Omega^-_k$, and since $|R(\Omega^+_k)| = 1$ and $\Omega^+_k = \Omega^-_k$ (by the induction hypothesis), it follows that $\mathcal{P}_k$ is the only releaser of $L$ at $\Omega^-_k$. Then from Part (d), $\Omega^+_k = \Omega^-_{k+1}$ 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.

**Claim 4.4.26.** If $I_2 = \emptyset$ and $R(I^-_0) = \emptyset$ and at time $I^-_0$, Sync1 = Sync2 = ⊥ and PawnSet is candidate-empty, then the following claims hold for all $i \in \{1, \ldots, \ell\}$:

(a) A PawnSet.reset() operation is not executed during $[I^-_0, \Omega^-_i]$.

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

(c) $\mathcal{P}_i$’s call to lock$\_\mathcal{P}_i()$ returns $\infty$, and $\mathcal{P}_i$ finishes lock$\_\mathcal{P}_i()$ during $T$, and Role[\mathcal{P}_i] = PAWN$P when $\mathcal{P}_i$’s call to lock$\_\mathcal{P}_i()$ 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 doPromote$\_\mathcal{P}_i()$.

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

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

(g) $t^\pi_{\mathcal{P}_i} < \Omega^-_i < t^{49}_{\mathcal{P}_i} < \Omega^+_i < t^{49}_{\mathcal{P}_i}$ and $\Omega^+_i \leq I^+_2$. 
(h) If $i = \ell$, then a PawnSet.reset() operation is not executed during $[I_0^-, \Omega_1^+]$.

(i) Throughout $[\gamma, \Omega_1^+]$, Sync1 = Sync2 = ⊥.

(j) If $\ell > 1$, $I_2^+ = \Omega_2^+ = \text{UNP}_{P_i}$.

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

(l) $R(I_2^+) = \emptyset$ 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 \( P_i \) at \( \Omega_i^- \). Then \( q \)’s PawnSet.promote() operation in line 65 returned value \( \langle P_i, s \rangle \), where \( s \in N \), and \( \Omega_i^- = \frac{\text{line 65}}{q} \). Then from the semantics of the PawnSet object it follows that the \( P_i \)-th entry of PawnSet was \( \langle \text{REG}, s \rangle = \langle 1, s \rangle \) immediately before \( \Omega_i^- \). Then from Claim 4.4.3(b) it follows that some process (say \( r \)) executed a PawnSet.collect(\( A \)) operation in line 55 where \( A[P_i] = s \). Then from the code structure, \( r \) read apply[\( P_i \)] = \( \langle \text{REG}, s \rangle \) in line 52. By Claim 4.4.4(a) apply[\( P_i \)] is set to value \( \langle \text{REG}, s \rangle \) only by process \( P_i \) when it executes a successful apply[\( P_i \)].CAS(\( \langle 1, 1 \rangle \), \( \langle \text{REG}, s \rangle \)), therefore \( P_i \) executed the same and broke out of the spin loop of line 2. Note that \( \Delta P_i < \Delta R < \Omega_i^- = \frac{\text{line 65}}{q} \).

Since Ctr = 0 throughout \( I_0 \), Ctr = 1 throughout \( I_1 \) and Ctr = 2 throughout \( I_2 \), it follows that Ctr is increased only at points \( I_1^- \) and \( I_2^- \) during \( T \). Since \( K \) and \( Q \) 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 \( K \) and \( Q \) increases the value of Ctr during \( T \). Since \( \Omega_i^- \geq \Omega_i^- = \gamma > I_2^- \) (by Claims 4.4.23(a) and 4.4.23(d) and 4.4.25(a)), \( P_i \) becomes a releaser of \( L \) only after \( I_2^- \) (the point at which \( Q \) became a releaser of \( L \)). Thus, \( P_i \) is not among the first two releasers of \( L \), thus \( P_i \notin \{ K, Q \} \). Then it follows that \( P_i \) does not increase Ctr. Therefore \( P_i \)’s Ctr.inc() operation in line 5 returns value \( 2 = \text{PAWN} \), and thus \( P_i \) sets Role[\( P_i \)] to \( 2 = \text{PAWN} \) in line 5. Then from the code structure \( P_i \) satisfies the if-condition of line 6 and proceeds to spin in line 7.

Case a - \( P_i \) receives a signal to abort while busy-waiting in line 7. Then \( P_i \) stops spinning in line 7 and executes abort._P_i(). Since \( P_i \) last set Role[\( P_i \)] to \( \text{PAWN} \) in line 5, it then follows from the code structure that \( P_i \) proceeds to execute lines 18-20 and satisfies the if-condition of line 20 and then executes a PawnSet.abort(\( P_i, s \)) 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 \( \text{PawnSet.abort}(\mathcal{P}_i, s) \) operation in line \( 21 \) during \([l_i^-, \Omega_i^+]\), thus \( \Omega_i^+ > \Omega_i^- \). Since \( \mathcal{P}_i \) has exclusive-registration access to \( \text{PawnSet} \) during \([\Omega_i^-, \Omega_i^+]\), and \( p \) has not executed any of its cease-release events or reset \( \text{PawnSet} \) during \([\mathcal{A}_i, \mathcal{B}_i]\), and \( \mathcal{A}_i < \Omega_i^- \), it then follows that \( \text{PawnSet} \) was not reset during \([\Omega_i^-, \mathcal{B}_i]\). Then since the \( \mathcal{P}_i \)-th entry of \( \text{PawnSet} \) was last changed to \( \langle \text{PRO}, s \rangle = \langle 2, s \rangle \) at \( \Omega_i^- \), it remains \( \langle \text{PRO}, s \rangle \) throughout \([\Omega_i^-, \mathcal{B}_i]\). Then \( \mathcal{P}_i \)'s \( \text{PawnSet.abort}(\mathcal{P}_i, s) \) operation in line \( 21 \) returns \texttt{false} by the semantics of the \( \text{PawnSet} \) object. Then \( p \) satisfies the if-condition of line \( 21 \) proceeds to set \( \text{Role}[\mathcal{P}_i] \) to \( \text{PAWN}_P \) in line \( 22 \) and then returns \( \infty \) from its call to \( \text{abort}_{\mathcal{P}_i}() \) and \( \text{lock}_{\mathcal{P}_i}() \).

**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 \( \Omega_i^- \) by executing a \( \text{PawnSet.promote}() \) 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 \( \langle j, \text{seq} \rangle = \langle \mathcal{P}_i, s \rangle \). Then \( q \) executes a \( \text{apply}[\mathcal{P}_i].\text{CAS}(\langle \text{REG}, s \rangle, \langle \text{PRO}, s \rangle) \) operation in line \( 70 \).

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

Since \( \mathcal{P}_i \) is busy-waiting in line \( 7 \) for \( \text{apply}[\mathcal{P}_i] \) to change to \( \langle \text{PRO}, s \rangle \), it then follows that \( \mathcal{P}_i \) busy-waits throughout \( (\Omega_i^-, \overline{\mathcal{A}}_q) \), and reads \( \text{apply}[\mathcal{P}_i] = \langle \text{PRO}, s \rangle \) when it executes line \( 7 \) for the first time after \( \overline{\mathcal{A}}_q \). Then \( \mathcal{P}_i \) breaks out of the spin loop, and then from
the code structure, $\mathcal{P}_i$ proceeds to set Role[$\mathcal{P}_i$] to 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 lock$_{\mathcal{P}_i}$() in line 17 with value $\infty$. Note that $\Omega_i^- < t_{\mathcal{P}_i}^i$.

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(d)), it follows from Claim 4.4.13 that $\mathcal{P}_i$ has exclusive write-access to objects Sync1 and Sync2 and exclusive registration-access to PawnSet throughout $[\Omega_i^{-}, \Omega_i^{+})$.

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

Since Role[$\mathcal{P}_i$] = PAWN_P when $\mathcal{P}_i$’s call to lock$_{\mathcal{P}_i}$() returns (by Part (c)), Role[$\mathcal{P}_i$] = PAWN_P at $t_{\mathcal{P}_i}^{34}$. Since 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 release$_{\mathcal{P}_i}$(), $\mathcal{P}_i$ only executes lines 34-35, 42 and 43-50. Then Figure 4.16 follows.

From an inspection of Figures 4.15 and 4.16, $\mathcal{P}_i$ does not execute a call to helpRelease$_{\mathcal{P}_i}$() or execute a Ctr.CAS(1,0) operation in line 36 during release$_{\mathcal{P}_i}$(). 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 doPromote$_{\mathcal{P}_i}$() 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 doPromote$_{\mathcal{P}_i}$(). Hence, Part (d) follows. Then $\Omega_i^+$ 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 Sync1 or Sync2 during lock$_{\mathcal{P}_i}$() and release$_{\mathcal{P}_i}$(). Therefore, $\mathcal{P}_i$ does not change Sync1 or Sync2 during $[\Omega_i^{-}, \Omega_i^{+})$.

Proof of Part (g) if Part (a) for $i$ is true: As argued in Part (b) and (c), $\Omega_i^{-} < \Omega_i^{-}$, and $\Omega_i^{-} < t_{\mathcal{P}_i}^{4}$. Since $t_{\mathcal{P}_i}^{34} < t_{\mathcal{P}_i}^{49}$ and $t_{\mathcal{P}_i}^{34} < t_{\mathcal{P}_i}^{41}$, it then follows that $t_{\mathcal{P}_i}^{34} < \Omega_i^{-} < t_{\mathcal{P}_i}^{41}$.

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 doPromoteP_i(). If cease-release event \( \theta_{P_i} \) occurs then \( \Omega_i^+ \) is the point when \( P_i \)'s cease-release event \( \theta_{P_i} \) occurs, i.e., \( \Omega_i^+ = t_{P_i}^{GS} \). Then \( P_i \) changes Ctr to 0 and the Ctr-cycle interval \( T \) ends at \( \Omega_i^+ = t_{P_i}^{GS} = I_2^+ \).

If cease-release event \( \pi_P \) occurs then \( \Omega_i^+ \) is the point when \( P_i \)'s cease-release event \( \pi_{P_i} \) occurs, i.e, \( \Omega_i^+ = t_{P_i}^{GS} < I_2^+ \).

Since \( P_i \) calls doPromoteP_i() only in line 46 (by inspection of Figure 4.16), it then follows that \( \Omega_i^+ \in \{t_{P_i}, t_{P_i}^{GS} \} \). 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 \( \pi_{P_i} \) and \( \theta_{P_i} \) occurs during \( P_i \)'s call to doPromoteP_i(). If cease-release event \( \theta_{P_i} \) occurs then \( \Omega_i^+ \) is the point when \( P_i \)'s cease-release event \( \theta_{P_i} \) occurs, i.e, \( \Omega_i^+ = t_{P_i}^{GS} \). Then \( P_i \) changes Ctr to 0 and the Ctr-cycle interval \( T \) ends at \( \Omega_i^+ = t_{P_i}^{GS} \), and thus \( \ell = i \). This is a contradiction to the assumption \( i = \ell \), hence \( P_i \)'s cease-release event \( \pi_{P_i} \) occurs during \( P_i \)'s call to doPromoteP_i(). Then \( \Omega_i^+ \) is the point when \( P_i \)'s cease-release event \( \pi_{P_i} \) occurs, i.e, \( \Omega_i^+ = t_{P_i}^{GS} \). From an inspection of Figures 4.15 and 4.16 and the code, it follows that \( P_i \) does not execute a PawnSet.reset() operation during \( [t_{P_i}, t_{P_i}^{GS}] \), and \( P_i \) calls doPromoteP_i() only in line 46. Since \( \Omega_i^+ = t_{P_i}^{GS} \), from an inspection of the code of doPromoteP_i(), \( P_i \) does not execute a PawnSet.reset() operation during \( [t_{P_i}^{GS}, t_{P_i}^{GS}] \). Then \( P_i \) does not execute a PawnSet.reset() operation during \( [\Omega_i^- , \Omega_i^+] \).

Since \( P_i \) is the only releaser of \( L \) throughout \( [\Omega_i^- , \Omega_i^+] \) (Claim 4.4.25(e)), it follows from Claim 4.4.13 that \( P_i \) 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 \( P_i \) 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), Sync1 = Sync2 = ⊥ at $\Omega_1^− = \gamma$. From Claims 4.4.25(a) and 4.4.25(d), it follows that $\gamma = \Omega_1^− < \Omega_1^+ = \Omega_2^− < \Omega_2^+ = \Omega_3^− \ldots < \Omega_{i-1}^+ = \Omega_i^− < \Omega_i^+$. From Claim 4.4.25(e), for all $t \in [\Omega_i^−, \Omega_i^+]$, $R(t) \in \{P_i\}$. Then $P_i$ has exclusive write-access to Sync1 and Sync2 throughout $[\Omega_i^−, \Omega_i^+]$. Since $P_i$ does not change Sync1 or Sync2 during $[\Omega_i^−, \Omega_i^+]$ (Part (i)), it then follows that Sync1 = Sync2 = ⊥ throughout $[\Omega_1^−, \Omega_i^+]$.

**Proof of (j):** As argued in Part (i), exactly one cease-release event among $\pi_P$ and $\theta_P$ occurs during $P_i$’s call to doPromoteP(). If cease-release event $\pi_P$ occurs then $P_i$ promotes some process, and thus the number of processes that get promoted during $T$ is larger than $\ell$, which contradicts the definition of $\ell$. Hence, cease-release event $\theta_P$ occurs during doPromoteP() and $\Omega_\ell^+$ is the point when cease-release event $\theta_P$ occurs, i.e., $\Omega_\ell^+ = \Omega_P^*$. Since Ctr is changed from 2 to 0 when $\theta_P$ occurs, the Ctr-cycle interval $T$ ends at $\Omega_\ell^+ = \Omega_P^*$, and thus $I_2^+ = \Omega_\ell^+ = \Omega_P^*$.

**Proof of (k) and (l):**

**Case a - $\ell = 0$:** Consider the first PawnSet.promote() operation at $\gamma$. Since $\ell = 0$, the PawnSet.promote() operation at $\gamma$ returns value $\langle \bot, \bot \rangle$. Then from Claim 4.4.23(j), it follows that $B$’s cease-release event $\theta_B$ occurs at $t' = \Omega_B^* = \gamma$, and throughout $[\gamma, t']$ no process is promoted, and for all $t \in [\gamma, t']$, $R(t) = \{B\}$. Since Ctr is changed from 2 to 0 when $\theta_B$ occurs, the Ctr-cycle interval $T$ ends at $t' = \Omega_B^*$, and thus $I_2^+ = \Omega_B^* = t'$. 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 $B$ executed a PawnSet.reset() operation in line 67 during $[\gamma, t']$, and thus PawnSet is candidate-empty immediately after. Since for all $t \in [\gamma, t')$, $R(t) = \{B\}$, 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 $t \in [\gamma, t')$, $R(t) = \{ B \}$, $B$ has exclusive write-access toSync1 and Sync2 throughout $[\gamma, t')$ (follows from Claim [4.4.13]). Since $\text{Sync1} = \text{Sync2} = \bot$ at $\gamma$ (Claim [4.4.23](l)), and $B$ does not write to Sync1 and Sync2 during $[\gamma, t']$, it follows that $\text{Sync1} = \text{Sync2} = \bot$ throughout $[\gamma, t'] = [\gamma, I^+_2]$.

**Case b -** $\ell \geq 1$ : From Part (i), $I^+_2 = \Omega^+_\ell = \{ \}$. Then from Part (ii), it follows that $\text{Sync1} = \text{Sync2} = \bot$ 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 $\Omega^+_\ell$, $R(I^+_2) = \emptyset$.

Since $\Omega^+_\ell = \{ \} = \{ \}$, $\mathcal{P}_i$ executed line 68 and before that line 67. Hence, $\mathcal{P}_\ell$ executed a PawnSet.reset() operation at $t^\mathcal{P}_\ell < \Omega^+_\ell$. Since $t^\mathcal{P}_\ell > t^\mathcal{P}_\ell^\mathcal{P}$ and $t^\mathcal{P}_\ell > \Omega^-_\ell$ (by Part (g)), it follows that $t^\mathcal{P}_\ell > \Omega^-_\ell$. Hence, $\mathcal{P}_\ell$ executed a PawnSet.reset() operation at $t^\mathcal{P}_\ell \in [\Omega^-_\ell, \Omega^+_\ell]$. Since $\mathcal{P}_\ell$ is the only releaser of $L$ throughout $[\Omega^-_\ell, \Omega^+_\ell]$ (Claim [4.4.25](c)), it follows from Claim [4.4.13] that $\mathcal{P}_\ell$ has exclusive registration-access to PawnSet throughout $[\Omega^-_\ell, \Omega^+_\ell$]). Then it follows that PawnSet is candidate-empty at $\Omega^-_\ell = I^+_2$. \[\]

**Claim 4.4.27.** $R(I^-_0) = \emptyset$ and at $I^-_0$, $\text{Sync1} = \text{Sync2} = \bot$ and 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 ($\text{Sync1} = \text{Sync2} = \bot$ and PawnSet is candidate-empty).

**Induction Step -** By the induction hypothesis, at $I^-_0$ for $T^{k-1}$, $R(I^-_0) = \emptyset$, and $\text{Sync1} = \text{Sync2} = \bot$ and 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 $\text{Sync1} = \text{Sync2} = \bot$ and 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 \( I_2 = \emptyset \). From Claim [4.4.21][f], it follows that \( \mathcal{K} \) is the only releaser of \( L \) during \( I_1 \). Since \( I_2 = \emptyset \), it then follows from Claim [4.4.21][c], that \( \mathcal{K} \)'s \( \text{Ctr.CAS}(1,0) \) operation in line [36] is successful, and the interval \( I_1 \) as well as \( T^{k-1} \) ends at time \( k_{\mathcal{K}} \). Then \( \mathcal{K} \)'s cease-release event \( \phi_{\mathcal{K}} \) occurs at \( k_{\mathcal{K}} = I^+_1 \), and thus there are no releasers of \( L \) immediately after \( T^{k-1} \) ends. And from Claim [4.4.21][g], it follows that \( \text{Sync}1 = \text{Sync}2 = \perp \) and \( \text{PawnSet} \) is candidate-empty when \( T^{k-1} \) ends.

**Case b - \( T^{k-1} \) ends at time \( I^+_2 \):** Then \( I_2 = \emptyset \). Then our proof obligation follows immediately from Claim [4.4.26][l].

Note that in the following claims, notations \( I_0, I_1, I_2, \lambda, \gamma, \Omega_i, \mathcal{K}, Q \) and \( \mathcal{P}_i \) are defined relative to a \( \text{Ctr} \)-cycle interval, as was defined previously in pages [106], [115] and [121]. The exact \( \text{Ctr} \)-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 \( L.\text{release()} \). From Claim [4.4.11][b], it follows that both \( p \) and \( q \) are releasers of \( L \) at \( t \). Consider the \( \text{Ctr} \)-cycle interval \( T \) such that \( t \in T \).

From Claim [4.4.27] it follows that at \( I^-_0 \), \( \text{Sync}1 = \text{Sync}2 = \perp \) and \( \text{PawnSet} \) is candidate-empty, and \( R(I^-_0) = \emptyset \). Then from Claims [4.4.21][a], [4.4.21][f], [4.4.23][a], [4.4.23][c] 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 \( t \in [I^-_2, \lambda) \), \( R(t) = \{\mathcal{K}, Q\} \). Then \( \{p, q\} = \{\mathcal{K}, Q\} \). Let \( p = \mathcal{K} \) and \( q = Q \) without loss of generality.

Recall that \( I^-_2 \) is the point in time when \( Q \) increases \( \text{Ctr} \) from 1 to 2 and sets \( \text{Role}[Q] \) to \( \text{QUEEN} \) in line [5]. Since \( q \)'s call to \( \text{lock}() \) returned a non-\( \perp \) value, it follows from an inspection of Figure 4.10 that \( Q \) returned either in line [17] or line [27]. Then \( Q \) either read a non-\( \perp \) value from \( \text{Sync}1 \) in line [14] or \( Q \) failed the \( \text{Sync}1.\text{CAS}(\perp, \infty) \) operation in line [26]. Since \( \text{Sync}1 = \perp \) at \( I^-_2 \) (by Claim [4.4.22][a]), and \( I^-_2 = \emptyset_\mathcal{Q} \), it then follows
that Sync1 is changed to a non-⊥ value during \([I_2^-, t]\). Clearly, \(Q\) does not change Sync1 during \([I_2^-, t]\).

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

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 \(\text{apply}[p]\) = \(\langle\text{REG}, s\rangle\) at \(I_0^-, t\), 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 \(I_0^-, Sync1 = Sync2 = ⊥\) and PawnSet is candidate-empty, and \(R(I_0^-) = \emptyset\). Then from Claim 4.4.23[k], it follows that exactly one call to doCollect() is executed during \(T\) by a process \(q \in \{K, Q\}\). Since processes are collected only during a call to doCollect(), \(q \in \{K, Q\}\) collects \(p\) during doCollect\(_q\)() during \(T\). And \(q\) does so by executing a PawnSet.collected\(_A\)() operation in line 55 where \(A[p] = s \in N\), and sets the \(p\)-th entry of PawnSet to \(\langle\text{REG}, s\rangle\). Since a PawnSet.promote() that returns \(\langle⊥, ⊥\rangle\) is executed at \(p\) during \(T\), it then follows from the semantics of the PawnSet object that \(p\) was promoted during \(T\). Then \(p = \mathcal{P}_i\), for some \(i \leq \ell\). Note that \(T\) does not end during \([Ω_i^−, Ω_i^+\]).

We now show that \(p\) is also notified of its promotion during \(T\). The process (say \(r\)) that promoted \(p\) by executing a PawnSet.promote() operation in line 55 also goes on to notify \(p\) of its promotion by executing a \(\text{apply}[p]\).CAS(\(\langle\text{REG}, s\rangle, \langle\text{PRO}, s\rangle\)) operation in
Since \( p \) does not abort, it follows from an inspection of Figure 4.15 and the code, that \( p \) spins on \( \text{apply}[p] \) in line 7 until its notification. Then \( p \) executes line 9 at \( t_p^1 < t_p^2 < t_p^5 = \Omega^- \). Since \( t_p^3 < \Omega_i^+ \) and \( T \) does not end before \( \Omega_i^+ \), it follows that \( p \) is notified during \( T \).

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

From Claim 4.4.27 it follows that at \( I_p^- \), \( \text{Sync1} = \text{Sync2} = \bot \) and \( \text{PawnSet} \) is candidate-empty, and \( \text{R}(I_p^-) = \emptyset \). Then from Claim 4.4.23(k), it follows that exactly one call to \( \text{docollect()} \) is executed during \( T \) by a process \( q \in \{K, Q\} \). Consider the point when \( q \) reads \( \text{apply}[p] \) in line 52. If \( q \) reads a value different from \( \langle \text{REG}, s \rangle \), then some process must have notified \( p \) during \( [t_p^2, t_q^2] \), and since \( I_p^- < t_p^2 \) and \( t_q^2 \in T \), our claim holds. If \( q \) reads the value \( \langle \text{REG}, s \rangle \) from \( \text{apply}[p] \), then \( q \) collects \( p \) during \( T \) by executing a \( \text{PawnSet}.\text{collect}(A) \) 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 \( \text{lock}_p() \) and returns a non-\( \bot \) value.

(b) \( p \) incurs \( \mathcal{O}(1) \) RMRs in expectation during its call to \( \text{lock}_p() \).
Proof. Proof of (a) and (b): From an inspection of the code 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}(). By assumption of the claim, p registered itself in line 2 by executing a successful apply[p].CAS(⟨⊥, ⊥⟩,⟨REG, s⟩) operation in line 2 and incurred \(\mathcal{O}(1)\) RMRs in the process. Then p proceeds to execute a Ctr.inc() operation in line 5 and stores the returned value in Role[p]. A Ctr.inc() operation returns values in \{KING, QUEEN, PAWN, ⊥\}. If it returns ⊥, p repeats the role-loop, and executes another Ctr.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 Ctr.inc() operation returns a non-⊥ value.

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

Case b - p executes a Ctr.inc() operation in line 5 that returns QUEEN. Then p increments Ctr from 1 to 2 in line 5 and sets Role[p] = QUEEN in line 5. Then from the code structure p proceeds to busy-wait on Sync1 in line 14. Since p increased Ctr from 1 to 2, \(\frac{A_{p}^{+}}{I_{p}^{-}} = I_{2}^{-}\) for some Ctr-cycle interval \(T\). From Claim 4.4.27 it follows that at \(I_{0}^{-}\), Sync1 = Sync2 = ⊥ and PawnSet is candidate-empty, and \(R(I_{0}^{-}) = \emptyset\). Then from Claim 4.4.22 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-⊥ value in line 17 and p does not change 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 \(\ell\) 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-⊥ value to Sync1 during \(T\) in line 37. Since Sync1 is written to only be a releaser of L, and \(\frac{A_{p}^{+}}{I_{p}^{-}} \in T\), it then follows that Sync1 is
changed to a non-$\perp$ value at most once during $T$. Then $p$ incurs at most one RMR while busy-waiting on $\text{Sync1}$. Hence, we have shown that Part (b) holds.

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

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

**Subcase (i)** - $\text{apply}[p] = \langle \text{REG}, s \rangle$ 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 $\text{apply}[p]$, and thus $\text{apply}[p]$ is changed from $\langle \text{REG}, s \rangle$ to $\langle \text{PRO}, s \rangle$ when $p$ is notified. Since $\text{apply}[p]$ is changed from $\langle \text{PRO}, s \rangle$ to some other value only by $p$, it then follows that $\text{apply}[p]$ remains $\langle \text{PRO}, s \rangle$ when $p$ reads $\text{apply}[p]$ for the first time after $p$ was notified. Then $p$ incurs one RMR when it reads $\text{apply}[p]$ 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 $\text{Role}[p] = \text{PAWN.P}$ in line 9 and proceeds to return $\infty$ in line 17. Then we have shown Parts (a) and (b) hold.

**Subcase (ii)** - $\text{apply}[p] = \langle \text{REG}, s \rangle$ at $I_0^−$ during $T$, for some $s \in \mathbb{N}$. Consider the only call to $\text{doCollect()}$ during $T$ by $q \in \{K, Q\}$. If $p$ registered itself (i.e., executed its $\text{apply}[p].\text{CAS}(\langle \perp, \perp \rangle, \langle \text{REG}, s \rangle)$ operation in line 2) before $q$ reads $\text{apply}[p]$ in line 52 during $\text{doCollect}_q()$, 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 $\text{apply}[p]$ during $T$. Then $p$ continues to busy-wait in line 7 until the $\text{Ctr}$-cycle interval $T$ ends and $\text{Ctr}$ is reset to 0.

If $\text{Ctr}$ is increased to 2 before $p$ reads $\text{Ctr}$ again in line 7, then let $T'$ be the $\text{Ctr}$-cycle
interval that starts when \( \text{Ctr} \) is reset to 0 at the end of \( T \). Since \( \text{apply}[p] \) was changed to a non-\( \langle \text{REG}, s \rangle \) value before the start of \( T' \), it follows that \( \text{apply}[p] = \langle \text{REG}, s \rangle \) 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 \( \text{Ctr} = 2 \) when \( p \) reads \( \text{Ctr} \) 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 \( \text{Ctr} \)-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 \( \text{Role}[p] = \{ \text{KING, QUEEN} \} \) in line 5 then Parts (a) and (b) hold as argued in Case a and Case b. If \( p \) sets \( \text{Role}[p] = \text{PAWN} \) in line 5 then it follows that \( \exists p \in T'' \), for some \( \text{Ctr} \)-cycle interval \( T'' \), such that \( \text{apply}[p] = \langle \text{REG}, s \rangle \) at \( I_0 \) for \( T'' \). Parts (a) and (b) hold as argued in Case c(i).

\[ \square \]

**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 \( \text{lock}_p() \) and returns a non-\( \bot \) value.

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

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

Consider \( p \)'s call to \( \text{lock}_p() \). Process \( p \) first attempts to register itself in line 2 by attempting to execute an \( \text{apply}[p].\text{CAS}(\langle \bot, \bot \rangle, \langle \text{REG}, s \rangle) \) operation. Now, \( \text{apply}[p] \) is changed from \( \langle \bot, \bot \rangle \) to a non-\( \langle \bot, \bot \rangle \) value only by \( p \) (Claim 4.4.4(a)). If \( \text{apply}[p] = \langle \bot, \bot \rangle \) at \( t_p \), then \( p \) executes a successful \( \text{apply}[p].\text{CAS}(\langle \bot, \bot \rangle, \langle \text{REG}, s \rangle) \) 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 \text{apply}[p] = \langle \bot, \bot \rangle$ at $t^{2}_{p}$, it follows that some process $p'$ executed a successful \text{apply}[p].\text{CAS}(\langle \bot, \bot \rangle, (\text{REG}, s')) in line 2 during \text{lock}_p(), and \text{apply}[p] = \langle \bot, \bot \rangle$ throughout $[t^{2}_{p}, t^{3}_{p}]$. Since calls to \text{lock}_p() are not executed concurrently, it follows that $p'$ has completed its call to \text{lock}_p() during $[t^{3}_{p}, t^{4}_{p}]$.

\textbf{Case 1 -} $p'$'s call to \text{lock}_p() returned $\bot$. Then it follows from the code structure that $p'$ executed a call to \text{abort}_p() and returned from line 18 or 33. Since $p$ executed a successful \text{apply}[p].\text{CAS}(\langle \bot, \bot \rangle, (\text{REG}, s')) 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 \text{true}, and thus $p$ could not have returned $\bot$ from line 18 during \text{abort}_p(). Then $p'$ returned $\bot$ in line 33 and thus $p'$ executed operations \text{apply}[p].\text{CAS}(\langle \text{REG}, s' \rangle, (\text{PRO}, s')) (in line 19), and \text{apply}[p].\text{CAS}(\langle \text{PRO}, s' \rangle, \langle \bot, \bot \rangle) (in line 32). Since, \text{apply}[p] can be changed from \langle \text{REG}, s' \rangle only to \langle \text{PRO}, s' \rangle, and from \langle \text{PRO}, s' \rangle only to \langle \bot, \bot \rangle, it then follows that $p'$ executes a successful \text{apply}[p].\text{CAS}(\langle \text{PRO}, s' \rangle, \langle \bot, \bot \rangle) (in line 32). Then $p'$ eventually resets \text{apply}[p] during its \text{lock}_p() call. Since \text{apply}[p] = \langle \bot, \bot \rangle throughout $[t^{3}_{p}, t^{4}_{p}]$ and $p'$ completed its call to \text{lock}_p() during $[t^{3}_{p}, t^{4}_{p}]$, we have a contradiction.

\textbf{Case 2 -} $p$'s call to \text{lock}_p() returned a non-$\bot$ value. Then from the code structure $p'$ executed operations \text{apply}[p].\text{CAS}(\langle \text{REG}, s' \rangle, \langle \text{PRO}, s' \rangle) (in line 16 or line 19) before returning from its call to \text{lock}_p(). Since \text{apply}[p] can be changed from \langle \text{REG}, s' \rangle only to \langle \text{PRO}, s' \rangle, and from \langle \text{PRO}, s' \rangle only to \langle \bot, \bot \rangle and only by a process with pseudo-ID $p$, it then follows that \text{apply}[p] = \langle \text{PRO}, s' \rangle when $p$'s \text{lock}_p() returns. Then it also follows that \text{apply}[p] = \langle \text{PRO}, s' \rangle until a process with pseudo-ID $p$ executes an \text{apply}[p].\text{CAS}(\langle \text{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 $\text{release}_p(j)$, for some integer $j$. Since a call to $\text{release}_p(j)$ is wait-free and all processes continue to take steps, it follows that eventually $r$ executes lines 48 and 49 where it reads value $(\text{PRO}, s')$ from $\text{apply}[p]$ in line 48 and resets $\text{apply}[p]$ with a $\text{apply}[p]\text{.CAS}((\text{PRO}, s'), (⊥, ⊥))$ operation in line 49. Since $p$ does not abort, and no other process calls $\text{lock}_p()$ concurrently, it then follows that eventually $p$ executes a successful $\text{apply}[p]\text{.CAS}((⊥, ⊥), (\text{REG}, s))$ operation in line 2. Since $\text{apply}[p]$ changed only once from $(\text{PRO}, s')$ to $(⊥, ⊥)$ while $p$ busy-waited in line 2 it follows that $p$ incurs $O(1)$ 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 $\text{abort}_p()$. If $p$’s call to $\text{abort}_p()$ returns $⊥$, then $p$’s passage ends, or else $p$’s $\text{lock}_p()$ returns non-$⊥$ value and $p$ calls $\text{release}_p()$ and $p$’s passage ends when the $\text{release}_p()$ method returns. Since $\text{abort}_p()$ and $\text{release}_p()$ 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 $\text{lock}_p()$ and all processes continue to take steps then $p$ eventually returns from $\text{lock}_p()$ with a non-$⊥$ value. Then $p$ eventually calls $\text{release}_p()$, and since $\text{release}_p()$ is wait-free, $p$ eventually completes its passage. If $p$ receives a signal to abort during $\text{lock}_p()$, 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(j) returns \texttt{true}, then there exists a concurrent call to \texttt{lock()} that eventually returns \texttt{j}.

Proof. The only operations that write a value to \texttt{Sync1} are \texttt{Sync1.CAS(\bot, \infty)} in line \ref{line:cas1} and \texttt{Sync1.CAS(\bot, j)} in line \ref{line:cas2}. From Claim 4.4.13, \texttt{Sync1} is written to only by a releaser of \texttt{L}. From Claim 4.4.27 it follows that at \( I_0^- \), \texttt{Sync1} = \texttt{Sync2} = \bot and \texttt{PawnSet} is candidate-empty, and \( R(I_0^-) = \emptyset \). Then from Claims 4.4.21(a), 4.4.21(b), 4.4.23(a), 4.4.23(c), 4.4.26(k), and 4.4.26(l), the only releasers of \texttt{L} during a \texttt{Ctr}-cycle interval \( T \), are \{\( \mathcal{K}, \mathcal{Q}, \mathcal{P}_1, \ldots, \mathcal{P}_t \}\}. 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 \texttt{Sync1} during \texttt{Ctr}-cycle interval \( T \).

Since \( p \) returns \texttt{true}, it then follows from an inspection of the code that \( p \) executed a successful \texttt{Sync1.CAS(\bot, j)} operation in line \ref{line:cas2} and thus failed the \texttt{Ctr.CAS(1, 0)} operation in line \ref{line:cas1} and \texttt{Role[p]} = \texttt{KING} at \ref{line:role}. Then \( p = \mathcal{K} \) for some \texttt{Ctr}-cycle interval \( T \). Since \( \mathcal{K} \) failed the \texttt{Ctr.CAS(1, 0)} operation in line \ref{line:cas1} it then follows that \texttt{Ctr} was increased to 1 by process \( \mathcal{Q} \) during \( T \), and \( I_2 = I_2^\mathcal{K} < I_2^\mathcal{Q} \). Since \( I_1^\mathcal{K} = I_1^\mathcal{Q} \) and \( I_1^\mathcal{K} < I_2^\mathcal{K} \), it then follows that \( \mathcal{Q} \)'s \texttt{lockQ()} call is concurrent to \( \mathcal{K} \)'s \texttt{releaseK()} call.

From Claim 4.4.27 it follows that at \( I_0^- \), \texttt{Sync1} = \texttt{Sync2} = \bot and \texttt{PawnSet} is candidate-empty, and \( R(I_0^-) = \emptyset \). Then from Claim 4.4.23(a), \texttt{Sync1} = \bot at \( I_2^- \), and \( \mathcal{K} \) and \( \mathcal{Q} \) are the only two releasers of \texttt{L} during \( [I_2^-, \lambda] \), where \( \lambda \) is the first point in time when \( T \) is changed to a non-\( \bot \) value, and \( \lambda = \min(\frac{\delta_0}{\mathcal{K}}, \frac{\delta_0}{\mathcal{Q}}) \).

Now, \texttt{Sync1} is reset only in line \ref{line:reset} and since \( \frac{\delta_1}{\mathcal{K}} > \frac{\delta_1}{\mathcal{Q}} \geq \lambda \) and \( \frac{\delta_2}{\mathcal{K}} > \frac{\delta_2}{\mathcal{Q}} \geq \lambda \), it then follows that \( \mathcal{K} \) and \( \mathcal{Q} \) do not reset \texttt{Sync1} during \( [I_2^-, \lambda] \). Since \( \mathcal{K} \) and \( \mathcal{Q} \) are the only processes with write-access to \texttt{Sync1}, \texttt{Sync1} is not reset during \( [I_2^-, \lambda] \).

Consider \( \mathcal{Q} \)'s \texttt{lock()} call (see Figure 4.10). Since \( \mathcal{K} \) executed a successful \texttt{Sync1.CAS(\bot, j)} operation and \texttt{Sync1} is not reset during \( [I_2^-, \lambda] \), it then follows that if \( \mathcal{Q} \) executes the \texttt{Sync1.CAS(\bot, \infty)} operation in line \ref{line:cas1} then the operation fails. From
an inspection of Figure 4.10, Q either returns from its lock() call in line 17 or line 27. In both these lines, Q returns the non-⊥ value stored in Sync1. Since K is the only process apart from Q that can write to Sync1, Q returns the value j that K wrote during its releaseK(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 O(n) CAS objects and read-write registers.

Now consider an implementation of object RandALockArray, where instance PawnSet is implemented using object SFMSUnivConst<AbortableProArrayn>, 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 I0−, Sync1 = Sync2 = ⊥ and PawnSet is candidate-empty, and R(I0−) = ∅. Then from Claims 4.4.21(a), 4.4.21(b), 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 [I2−, λ) for some Ctr-cycle interval T. More specifically, there are two releasers of L only during [I2−, λ), and the releasers are K and Q. From Claim 4.4.23[k] it follows that a doCollect() is executed only by K or Q but not both. Then it follows immediately that lines of doCollect() are not executed concurrently.

Since λ = min(λK, λQ), it follows from an inspection of Figures 4.8, 4.9, 4.10, 4.11 and
Lemma 4.4.7. (a) $\text{helpRelease}_p()$ and $\text{doPromote}_p()$ have an RMR complexity of $O(1)$.

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

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

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

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

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

If a call to `release_p(j)` returns `true`, then $p$ does execute a call to `doCollect_p()` in line 38, else it does not. Then from an inspection of `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 `SFMSUnivConst(AbortableProArray_n)`, 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 $O(n)$ RMR complexity.

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

(d) If a process’ call to `release(j)` returns `false`, then it incurs $O(1)$ RMRs during the call, otherwise it incurs $O(n)$ RMRs during the call.

The object requires $O(n)$ 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 \texttt{RandALockTree}, where \( N \) is the maximum number of processes that can access the lock concurrently. Object \texttt{RandALockTree} is a tree based randomized implementation of type \texttt{AbortableLock}. An algorithm that accesses an instance of object \texttt{RandALockTree} must satisfy the following:

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

(b) Methods \texttt{lock}_p() and \texttt{release}_p() 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 \( \Delta \). Then it follows that \( \Delta = \Theta(\log N / \log \log N) \). An execution of an algorithm that accesses an instance of an object \texttt{RandALockTree} where Condition \textbf{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().

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

4.5.1 High Level Description

We now describe the implementation of object \texttt{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 $\Delta$-ary tree $T$ of height $\Delta$ with $N$ leaves, called the arbitration tree. The internal nodes of the arbitration tree $T$ 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 $\text{leaf}_p$ in the tree. We say that a node is at level $i$ if its height is $i$, where the root has height $\Delta$ and the leaves of the tree have height 0.

Let $\text{path}_p$ denote the path from $\text{leaf}_p$ up to $\text{root}$, and $h_u$ denote the height of node $u$. We assume that the tree structure $T$ provides a function $\text{nodeOnPath}()$, such that, for a leaf node $\text{leaf}$ and integer $\ell$, the function $\text{nodeOnPath}(\text{leaf}, \ell)$ returns a pair $(u, i)$, where $u$ is the $\ell$-th node on the path from $\text{leaf}$ 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 $\text{path}_p$ of some process $p$. Process $p$ is said to capture lock $u.L$ if its call to $u.L\text{lock}()$ returns a non-$\perp$ value. Process $p$ is said to release lock $u.L$ if $p$ executes a call to $u.L\text{release}()$. Process $p$ is said to hand over all nodes from node $v$ to $u$ on $\text{path}_p$ to a process $q$, if $p$ executes a $v.L\text{release}(j)$ call that returns true, where $j = h_u \geq h_v$ and $q$ executes a concurrent $v.L\text{lock}()$ call that returns $j$. Process $p$ hands over node $u$ so that some other process can call $u.L\text{release}()$ on $p$’s behalf. Recall that method $\text{release}()$ of object RandALockArray has $O(1)$ RMR complexity if the method returns false, and $O(\Delta)$ RMR complexity if
Class RandALockTree

```plaintext
define Node: struct {
    L: RandALockArray
}

shared:
    T: complete $\Delta$-ary tree of height $\Delta$ and node type Node

local:
    v: Node init ⊥,
    i, ℓ, k: int init 0,
    abort_signal: boolean init false,
```

Method lock$_p$():

```plaintext
while ℓ < T.height do
    (v, i) ← T.nodeOnPath(leaf$_p$, ℓ + 1)
    val ← v.L.lock$_i()$
    if val = ∞ then ℓ ← ℓ + 1
    if val ∉ {⊥, ∞} then ℓ ← val
    if abort_signal = true then
        release$_p()$
        return ⊥
    end
end
return ∞
```

Method release$_p$():

```plaintext
while k ≤ ℓ do
    (v, i) ← T.nodeOnPath(leaf$_p$, k)
    if v.L.release$_i(\ell)$ then break
    k ← 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 RandALockTree to $O(\Delta)$ RMR complexity. Process $p$ starts to own node $u$ when $p$ captures $u.L$ 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 $u.L$ 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 $\text{owner}_u$.

**Lock capture protocol - lock$_p()$.** The lock capture protocol executed during lock$_p()$ is as follows. Process $p$ climbs up the tree on its path starting from leaf node leaf$_p$ to the root node of $T$, 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 $u.L.lock()$. If $p$'s $u.L.lock()$ call returns $\infty$ 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 $h_v = j$. We ensure that $j \geq h_u$.

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

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

Let $s$ be the height of the highest node $p$ owns. During release$_p()$, $p$ climbs up
\( \mathcal{T} \) and calls \( u.\text{L.release}_p(s) \) at every node \( u \) that it owns, until a call returns \textbf{true}. If a \( u.\text{L.release}_p(s) \) call returns \textbf{false}, then \( p \) is said to have released lock \( u.\text{L} \) (and therefore released node \( u \)), and thus \( p \) continues on its path. If a \( u.\text{L.release}_p(s) \) call returns \textbf{true}, then \( p \) has handed over all remaining nodes that it owns to some process that is executing a concurrent \( u.\text{L.lock}() \) 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 \textbf{path}_q. There is however a side effect of this strategy which is as follows: Suppose \( p \) owns nodes \( v \) and \( u \) on \textbf{path}_p such that \( (u, i) = \text{nodeOnPath}(\text{leaf}_p, h_u) \) and \( v \) is the \( i \)-th child on node \( u \). Now suppose \( p \) releases lock \( v.\text{L} \) at node \( v \). Since the lock at node \( v \) is now released, some process \( r = p \) may now capture lock \( v.\text{L} \) and then proceed to call \( u.\text{L.lock}_r() \). If process \( p \) has not yet released \( u.\text{L} \) by completing its call to \( u.\text{L.release}_i() \), then we have a situation where a call to \( u.\text{L.lock}_i() \) is made before a call to \( u.\text{L.release}_i() \) is completed. Since there can be at most one owner of lock \( v.\text{L} \), there can be at most one such call to \( u.\text{L.lock}_i() \) concurrent to \( u.\text{L.release}_i() \). This is precisely the reason why we designed object \textbf{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 \textbf{RandALockTree} (see Figure 4.17).

**Description of the \textit{lock}_p() method.** Suppose process \( p \) executes a call to \( \text{lock}_p() \). With every iteration of the while-loop, process \( p \) captures at least one node on its path from \( \text{leaf}_p \) to \( \mathcal{T}.\text{root} \). 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 $\text{path}_p$ and the index (say $r$) of $u$’s child node that lies on $\text{path}_p$, 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 $u.L$, and thus node $u$ by executing a call to $u.L\text{lock}()$ with pseudo-ID $r$. If $p$'s $u.L\text{lock}()$ returns an integer value (say $j$) then $p$ has been transferred all nodes on its path up to height $j$ (we ensure $j \geq h_u$). If $p$’s $u.L\text{lock}()$ returns $\infty$ then $p$ has captured lock $u.L$. In lines 4 and 5 $p$ stores the height of the highest captured node in its local variable $\ell_p$. 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 $\text{release}_p()$ in line 7 and then returns from its call to $\text{lock}_p()$ 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 $\infty$ in line 11 to indicate a successful $\text{lock}()$ call.

**Description of the $\text{release}_p()$ method.** Suppose process $p$ executes a call to $\text{release}_p()$. Let $s$ be the highest node $p$ owns at the beginning of $\text{release}_p()$. We later prove that $h_s = \ell_p$. During an iteration of the while-loop (lines 12-16), process $p$ either releases a node on its path from $\text{leaf}_p$ 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 $t \leq h_s$. In line 13 process $p$ determines the $t$-th node (say $u$) on $\text{path}_p$ and the index (say $r$) of $u$’s child node that lies on $\text{path}_p$, and stores them in local variables $v_p$ and $i_p$. In line 14 process $p$ releases $u.L$, and thus node $u$, by executing a call to $u.L\text{release}(h_u)$ with pseudo-ID $r$. If $p$’s $u.L\text{release}(h_u)$ returns $\text{false}$ then $p$ has successfully released lock $u.L$, and thus node $u$. If $p$’s $u.L\text{release}(h_u)$ returns $\text{true}$ then $p$ has successfully handed over all nodes from $u$ to $s$ on $\text{path}_p$ to some process that is executing a concurrent call to $u.L\text{lock}()$. 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 $\text{release}_p()$. 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 \texttt{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 \texttt{RandALockTree} where Condition 4.5.1 is satisfied. Consider an arbitrary node \( u \) on the tree \( T \). 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 \texttt{u.L.release(j)} call that returns \texttt{true}, where \( j \geq h_u > h_v \) and \( q \) executes a concurrent \texttt{u.L.lock()} call that returns \( j \). Process \( p \) is said to start to own node \( u \) when \( p \) captures \( u.L \) 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 \( u.L \), 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 \texttt{u.L.lock()} operation that returns value \( j \not\in \{\bot, \infty\} \), then \( j \geq h_u \).

(b) The value of \( \ell_p \) 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 \texttt{RandALockArray} (Theorem 4.4.2), it follows that some process (say \( q \)) executed a concurrent \texttt{u.L.release(j)} operation. Then from the code structure, \( q \) executed a \texttt{u.L.release(j)} in line 14 where \( \ell_q = j \). Then \( q \) also executed a \texttt{T.nodeOnPath(leaf_q, k)} operation in line 13 that returned \( \langle u, i \rangle \), for some \( i \), such that \( h_u = k_q \) (from the semantics of the \texttt{nodeOnPath()} method). Since \( j = \ell_q \geq k_q = h_u \), our claim follows.

**Proof of (b):** Process \( p \) writes to its local variable \( \ell_p \) only in lines 4 and 5. Clearly, \( p \) increases \( \ell_p \) every time it executes line 4. Now, suppose \( p \) executes line 5 where it writes
the value of \( \text{val}_p \) to \( \ell_p \), where \( v_p = u \), for some node \( u \). Since \( p \) satisfies the if-condition of line 5 and the \texttt{RandALockArray} method \texttt{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), \( \text{val}_p \geq h_u \). Since \( p \) also executed a \( T.nodeOnPath(\text{leaf}_p, b) \) operation in line 2, where \( b = \ell_p + 1 \) that returned \langle u, i \rangle, for some \( i \), such that \( h_u = b \) (from the semantics of the \( \text{nodeOnPath()} \) method), it follows that \( \text{val}_p \geq h_u = \ell_p + 1 \). Then, \( p \) increases \( \ell_p \) when \( p \) writes \( \text{val}_p \) to \( \ell_p \) in line 5.

**Proof of (c):** Let \( t' \) be the point in time such that \( p \) writes to its local variable \( \ell_p \) for the \( i \)-th time. We prove our claim by induction over \( i \).

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

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

**Case a -** \( p \) writes to \( \ell_p \) in line 4. Then \( p \) increased \( \ell_p \) 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 \( \text{path}_p \). 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 \( \infty \), where \( v_p = u \). Since \( h_u = j + 1 \), and \( p \) successfully captured lock \( u.L \), it follows that \( p \) owns the \( j + 1 \)-th node on \( \text{path}_p \).

**Case b -** \( p \) writes to \( \ell_p \) in line 5. Let \( \text{val}_p = x \) when \( p \) writes to \( \ell_p \) in line 5. From Part (b), it follows that \( \ell_p \) is increased every time it is written to, and therefore \( \text{val}_p = x > \ell_p \) when \( p \) writes to \( \ell_p \) in line 5. Thus, to prove our claim we need to show
that \( p \) owns all nodes on \( \text{path}_p \) with heights in the range \( \{j, \ldots, x\} \). Since \( p \) satisfies the if-condition of line 5 and the \texttt{RandALockArray} method \texttt{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. Thus, \( p \) has captured \( u.L \) and now owns node \( u \). It also follows that \( p \) has been handed over all nodes on \( \text{path}_p \) with heights in the range \( \{h_u + 1, \ldots, x\} \). Since \( h_u = j \), our claim follows.

A process is said to attempt to capture node \( u \) if it executes a \( u.L.lock() \) method in line 3.

\begin{claim}
\begin{enumerate}[(a)]
\item If two distinct processes \( p \) and \( q \) attempt to capture node \( v \), then their local variables \( i \) have different values.
\item A node has at most one owner at any point in time.
\end{enumerate}
\end{claim}

\begin{proof}
We prove our claims for all nodes of height at most \( h \), by induction over integer \( h \).

\textbf{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 \texttt{nodeOnPath(\langle \text{leaf}_p, 1 \rangle)} and \texttt{nodeOnPath(\langle \text{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, \( \text{leaf}_p \) and \( \text{leaf}_q \) are distinct leaf nodes of tree \( T \), and thus from the semantics of the \texttt{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 \( u.L.lock() \) for the same \( i \), and thus it follows from the mutual exclusion property of object \texttt{RandALockArray}, that at most one process captures \( u.L \). By definition, a process can become an owner of node \( u \) only if it captures \( u.L \) 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 \texttt{nodeOnPath}(\langle \text{leaf}_p, h \rangle) and \texttt{nodeOnPath}(\langle \text{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 \texttt{nodeOnPath()} method, \( i = j \) only if the \((h - 1)\)-th nodes on \texttt{path}_p and \texttt{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 \texttt{path}_p and \texttt{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 \texttt{lock()} call. Then both processes executed line 11 and thus \( \ell_p = \ell_q = T.height \) holds. Then from Claim 4.5.1(c) it follows that both \( p \) and \( q \) own node \( T.root \). But from Claim 4.5.2(b), at most one process may own \( T.root \) at any point in time – a contradiction. □

**Claim 4.5.3.** Process \( p \) repeats the while-loop in \texttt{lock()} at most \( \Delta \) times.

**Proof.** Consider an arbitrary process \( p \) that calls \texttt{lock()} . From the code structure of \texttt{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 \( \ell_p \) every time it repeats an iteration of the while-loop. Since the height of the \( T \) is \( \Delta \), 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 $\Delta$ 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. $\square$

Lemma 4.5.3. Process $p$ incurs $O(\Delta)$ RMRs during $release_p()$.

Proof. Consider $p$’s call to $release()$. Since $\ell_p \leq T.height = \Delta$, it follows from an inspection of the code that during $release()$, $p$ executes at most $\Delta$ 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 $O(\Delta)$ RMRs during a call to $L.release()$, if the call returns $true$, otherwise $O(1)$ RMRs. Then our claim follows immediately. $\square$

Lemma 4.5.4. Process $p$ incurs $O(\Delta)$ 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 $O(1)$ RMRs in expectation during the call. Since $p$ repeats the while-loop in $lock()$ at most $\Delta$ times (by Claim 4.5.3), and $p$ does not receive a signal to abort during $lock_p()$, it follows that $p$ incurs $O(\Delta)$ RMRs in expectation during $lock_p()$. 
Case b - $p$ receives a signal to abort during $\text{lock}_p()$. As per the properties of object $\text{RandALockArray}$ (Theorem 4.4.2), if a process aborts during a call to $\text{L.lock}()$, then the process incurs $O(\Delta)$ RMRs in expectation during the call. Since $p$ repeats the while-loop in $\text{lock}()$ at most $\Delta$ times (by Claim 4.5.3), and $p$ executes at most one call to $\text{u.L.lock}()$ after having received an abort signal, it follows that $p$ incurs $O(\Delta)$ RMRs in expectation during $\text{lock}_p()$.

\begin{lemma}
Method $\text{release()}$ is wait-free.
\end{lemma}

\begin{proof}
As per the bounded exit of object $\text{RandALockArray}$, method $\text{release()}$ of the object is wait-free. Then our claim follows immediately from an inspection of the code of $\text{release()}$.
\end{proof}

\begin{lemma}
The abort-way is wait-free and has $O(\Delta)$ RMR complexity.
\end{lemma}

\begin{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 $\text{release}_p()$ is wait-free, and has $O(\Delta)$ RMR complexity. From Claim 4.5.3 a process repeats the while-loop in $\text{lock}_p()$ at most $\Delta$ 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 $\text{u.L.lock}()$ in line 3 and that a process incurs at most $O(\Delta)$ 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 $\text{u.L.lock}()$ in line 3, for some node $u$, then the process executes $\text{u.L.lock}()$ in a wait-free manner and incurs $O(\Delta)$ RMR during the call, and does not call $\text{v.L.lock}()$ for any other node $v$.

Suppose that $p$ has received a signal to abort, and $p$ executes a call to $\text{u.L.lock}()$ 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 $\text{RandALockArray}$ (Theorem 4.4.2), its
abort-way is wait-free and has $O(\Delta)$ RMR complexity. Then $p$ executes the $u.L.lock()$ call in line 3 in a wait-free manner and incurs $O(\Delta)$ RMR complexity. It then goes on to satisfy the if-condition of line 6 and executes a call to $\text{release()}$ in line 7 and returns $\bot$ 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 $O(\log N/\log \log N)$, in expectation against the weak adversary. The algorithm requires $O(N)$ CAS objects and read-write registers.
Chapter 5

Conclusion

We presented a randomized abortable mutual exclusion algorithm with $O(\log N / \log \log N)$ 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 *RandALockArray* uses sequence numbers to solve the classic ABA problem, and therefore the registers of the *apply* array are unbounded. Our universal construction object *SFMSUnivConst()* 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 *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 *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 $\text{lock}_p()$ during a passage. The value of $\text{Role}[p]$ at various times is as follows.

<table>
<thead>
<tr>
<th>Points in time</th>
<th>Value of $\text{Role}[p]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_5^p$</td>
<td>${\infty, \text{KING}, \text{QUEEN}, \text{PAWN}}$</td>
</tr>
<tr>
<td>$[t_7^p, t_8^p]$</td>
<td>PAWN</td>
</tr>
<tr>
<td>$t_9^p$</td>
<td>PAWN</td>
</tr>
<tr>
<td>$t_{13}^p$</td>
<td>${\text{KING}, \text{QUEEN}, \text{PAWN}_P}$</td>
</tr>
<tr>
<td>$t_{14}^p$</td>
<td>QUEEN</td>
</tr>
<tr>
<td>$[t_{16}^p, t_{17}^p]$</td>
<td>${\text{KING}, \text{QUEEN}, \text{PAWN}_P}$</td>
</tr>
</tbody>
</table>

Proof. Since the values returned by a $\text{Ctr.inc()}$ operation are in $\{\infty, 0, 1, 2\} = \{\infty, \text{KING}, \text{QUEEN}, \text{PAWN}\}$, $\text{Role}[p]$ is set to one of these values in line 5. Hence, $\text{Role}[p] \in \{\infty, \text{KING}, \text{QUEEN}, \text{PAWN}\}$ at $t_5^p$. If $p$ satisfies the if-condition of line 6, then $\text{Role}[p] = \text{PAWN}$, and $p$ changes $\text{Role}[p]$ next only in line 9. Hence, $\text{Role}[p] = \text{PAWN}$ during $[t_7^p, t_8^p]$. In line 9, $p$ changes $\text{Role}[p]$ to $\text{PAWN}_P$ and does not change $\text{Role}[p]$ thereafter. Hence, $\text{Role}[p] = \text{PAWN}_P$ at $t_9^p$.

Process $p$ does not change $\text{Role}[p]$ after line 9. To break out of the getLock loop, $\text{Role}[p] \in \{\text{KING}, \text{QUEEN}, \text{PAWN}_P\}$ must be satisfied when $p$ executes line 12. Hence, $\text{Role}[p] = \{\text{KING}, \text{QUEEN}, \text{PAWN}_P\}$ during $[t_{10}^p, t_{14}^p]$. Since $p$ executes line 13 only after breaking out of the getLock loop, $\text{Role}[p] \in \{\text{KING}, \text{QUEEN}, \text{PAWN}_P\}$ at $t_{13}^p$. If $p$ satisfies the if-condition of line 13, then $\text{Role}[p] = \text{QUEEN}$, and since $p$ does not change $\text{Role}[p]$ thereafter, $\text{Role}[p] = \text{QUEEN}$ at $t_{14}^p$.

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

<table>
<thead>
<tr>
<th>Points in time</th>
<th>Value of Role[p]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[t_{19}^p, t_{20}^p]</td>
<td>{QUEEN, PAWN}</td>
</tr>
<tr>
<td>t_{21}^p</td>
<td>PAWN</td>
</tr>
<tr>
<td>[t_{22}^p, t_{23}^p]</td>
<td>PAWN_P</td>
</tr>
<tr>
<td>[t_{26}^p, t_{30}^p]</td>
<td>QUEEN</td>
</tr>
</tbody>
</table>

Proof. Process $p$ calls $\text{abort}_p()$ 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 $\text{abort}_p()$ is line 2, 7, or line 14. From Claim A.0.4, it follows that $\text{Role}[p] = \text{PAWN}$ at $t_{19}^p$, and $\text{Role}[p] = \text{QUEEN}$ at $t_{21}^p$.

Now, $p$'s local variable $\text{flag}$ is set to value $\text{true}$ 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 $\text{abort}_p()$. Hence, $\text{Role}[p] \in \{\text{PAWN, QUEEN}\}$ in $[t_{19}^p, t_{20}^p]$, since $p$ changes $\text{Role}[p]$ next only in line 22.

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

\[ \square \]

Claim A.0.6. Suppose a process $p$ executes a call to $\text{release}_p(j)$ during a passage. The value of $\text{Role}[p]$ at various points in time is as follows.
<table>
<thead>
<tr>
<th>Points in time</th>
<th>Value of Role[$p$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t^{34}_p$, $t^{35}_p$</td>
<td>{KING, QUEEN, PAWN.P}</td>
</tr>
<tr>
<td>$t^{40}_p$, $t^{40}_p$</td>
<td>KING</td>
</tr>
<tr>
<td>$t^{43}_p$</td>
<td>QUEEN</td>
</tr>
<tr>
<td>$t^{46}_p$</td>
<td>PAWN.P</td>
</tr>
<tr>
<td>$t^{49}_p$, $t^{50}_p$</td>
<td>{KING, QUEEN, PAWN.P}</td>
</tr>
</tbody>
</table>

**Proof.** Suppose the point in time $t^{34}_p$. Then, $p$ is executing a call to $\text{release}_p(j)$, and $p$ last executed a call to $\text{lock}_p()$ that returned a non-$\bot$ value. Then, $p$’s call to $\text{lock}_p()$ either returned from line $17$ in $\text{lock}_p()$ or from line $23$ or line $27$ in $\text{abort}_p()$. From Claim A.0.4, Role[$p$] $\in$ {KING, QUEEN, PAWN.P} at time $t^{44}_p$ and from Claim A.0.5, Role[$p$] = PAWN.P at $t^{23}_p$ and Role[$p$] = QUEEN at $t^{27}_p$. Therefore, Role[$p$] $\in$ {KING, QUEEN, PAWN.P} at time $t^{44}_p$.

From Claim A.4.2[b], Role[$p$] is unchanged during $\text{release}_p()$. Therefore, Role[$p$] $\in$ {KING, QUEEN, PAWN.P} during $[t^{44}_p, t^{45}_p]$ and $[t^{40}_p, t^{49}_p]$. Then, from the if-conditions of lines $35$, $42$ and $45$, it follows immediately that Role[$p$] = KING during $[t^{44}_p, t^{45}_p]$, and Role[$p$] = QUEEN at $t^{43}_p$, and Role[$p$] = PAWN.P at $t^{46}_p$.

$\Box$

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

<table>
<thead>
<tr>
<th>Points in time</th>
<th>Value of Role[$p$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t^{51}_p$, $t^{59}_p$</td>
<td>{KING, QUEEN}</td>
</tr>
<tr>
<td>$t^{65}_p$, $t^{68}_p$</td>
<td>{KING, QUEEN}</td>
</tr>
<tr>
<td>$t^{65}_p$, $t^{71}_p$</td>
<td>{KING, QUEEN, PAWN.P}</td>
</tr>
</tbody>
</table>

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


From a code inspection, \texttt{doCollect}_p() is called by \textit{p} only in lines 29 and 38. From Claim \textbf{A.0.5} \text{Role}[p] = \text{QUEEN} at \(t^{	ext{29}}_p\) and from Claim \textbf{A.0.6} \text{Role}[p] = \text{KING} at \(t^{	ext{38}}_p\). Since \text{Role}[p] is unchanged during \texttt{doCollect}_p(), it follows that \text{Role}[p] \in \{\text{KING, QUEEN}\} during \([t^{	ext{29}}_p, t^{	ext{38}}_p]\).

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

Now, suppose \(p\) executes a call \texttt{doPromote}_p(). From a code inspection, \texttt{doPromote}_p() is called by \textit{p} only in lines 46 and 62. From Claim \textbf{A.0.6} \text{Role}[p] = \text{PAWN}_P at \(t^{	ext{46}}_p\) and from earlier in this claim, \text{Role}[p] \in \{\text{KING, QUEEN}\} at \(t^{	ext{62}}_p\). Since \text{Role}[p] is unchanged during \texttt{doPromote}(), it follows that \text{Role}[p] \in \{\text{KING, QUEEN, PAWN}_P\} during \([t^{	ext{46}}_p, t^{	ext{62}}_p]\). \qed
Bibliography


