2018-04-26

Efficient Shared Memory Algorithms for Bounding Space

Aghazadeh, Zahra

UNIVERSITY OF CALGARY

Efficient Shared Memory Algorithms for Bounding Space

by

Zahra Aghazadeh

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

April, 2018

Abstract

It is the state of the art that computer systems use multi-core architectures. In order to exploit the benefits of parallelism, multiple processors have to cooperate and concurrently communicate by executing operations on shared objects. It is challenging in shared memory algorithms to deal with the inherent asynchrony of processes. To overcome this difficulty and to achieve high efficiency, algorithm designers often assume unbounded space. This work focuses on designing time-efficient shared memory algorithms while avoiding unbounded space.

One example is that of making shared objects writable: Many standard primitives (for instance, compare-and-swap or fetch-and-add) do not provide a `Write()` operation that unconditionally changes the object's state to the one provided as a parameter. Adding `Write()` operations without sacrificing efficiency is challenging if space is limited. We provide a space-efficient solution for making synchronization primitives writable with optimal step complexity. A special case of making an object writable is to augment non-resettable objects with `Reset()` operations. We show how our general transformation can be improved to achieve optimal space implementations of long-lived test-and-set objects with time-efficient `Reset()` operations.

Another example concerns the ABA problem: Even though a process retrieves the same value twice in a row from a shared object, it is still possible that the value of the object has changed multiple times. We investigate the time and space complexity of detecting ABAs in shared memory algorithms for systems with bounded base objects. To that end, we propose a new primitive called an ABA-detecting register, and we give an efficient implementation of this type using asymptotically optimal number of bounded registers.

Finally we deal with a more general unbounded space problem: Many applications employ the tagging technique, where shared objects get augmented with additional values, called tags. Unbounded tags are mainly used in those applications, because bounding them is too complicated and error prone. We introduce a new primitive that provides an abstraction for avoiding

unbounded tags. Also, we propose optimally time-efficient implementations of this primitive from bounded objects. In addition to straightforward applications that use tags directly, our implementations can also often be used for memory reclamation.

Table of Contents

List of Figures and Illustrations

Chapter 1

Introduction

With the exponential growth of information, it is now more than ever necessary to benefit from the power of parallelism offered by distributed computing. As powerful as these systems can be, designing an efficient distributed system that exploits the parallelism can be a tremendously difficult job. The main challenges are dealing with asynchrony that is inherent in hardware, and failures of processes.

In a shared memory system, processes communicate by executing operations on shared objects, either provided by the hardware, or by a software layer. These shared objects allow programmers to decompose larger tasks into smaller modules. Hence, having efficient implementations of tasks that keep appearing in distributed algorithms is desirable. To achieve this, researchers have been approaching this problem from different directions, for instance by

- developing more efficient implementations of known primitives,
- investigating the possibility and the complexity of implementing some primitives from others,
- introducing new and meaningful primitives that can be used as building blocks for distributed algorithms, and
- proposing universal transformations of primitives into new ones with enhanced functionality or efficiency.

To overcome the challenges of implementing powerful shared objects, algorithm designers often assume unbounded space, that is they use either an unbounded number of objects, or objects that can store unbounded values.

Objectives. The goal of this thesis is to introduce new primitives and new general transformations using bounded space for common problems that have trivial solutions only if unbounded space is available. Although obtaining such general transformations is intricate, once designed and proved correct, they can be applied easily in a variety of contexts, provided that they work in a "black box" fashion. This way the transformation builds a layer around the original input implementation rather than altering its delicate inner mechanisms.

Writable Objects. The first problem that we consider is a general transformation to make an object *writable*, that is augmenting the object with a `Write()` operation. A `Write(`x`)` changes the value of the object to x regardless of the state of the object. A `Reset()` operation is a special case of a `Write()`, which assigns an initial value to the implemented object. A simple transformation that achieves this goal maintains an unbounded pool of objects of the input type, as well as a pointer that identifies among these a designated "current" object. The `Reset()` operation updates (or "swings") this pointer to a new object in the pool, and all other operations are applied on the designated object by first following the pointer. This transformation, while simple and wait-free, uses unbounded space and is neither practical nor theoretically interesting.

In Chapter 4, we provide a transformation that makes any object writable provided that it is possible to write to the object non-concurrently, while preserving bounded space and wait-freedom. Our transformation uses a polynomially many (in number of processes) copies of the original object and registers, and only increases the step complexity of operations on the object by a constant additive term.

This transformation can be employed to augment existing interesting objects, such as compare-and-swap, and fetch-and-add, with a `Write()` or a `Reset()` operation. Another immediate application of this transformation is an implementation, with optimal step complexity, of a k-word register from $O(kn^2)$ single-word registers, where n (throughout this thesis) represents the number of processes.

More Efficient Long-Lived Test-And-Set. The specific characteristics of one particular object, called test-and-set (TAS), allow us to achieve better transformations than the general transformation of Chapter 4, in terms of step and space complexities. Test-and-set objects are standard synchronization primitives that can, for example, be used to protect a critical section in a mutual exclusion algorithm. This object stores a bit, which is initially 0. The one-time version allows only one operation, `TAS()`, which atomically returns the value of the bit and sets it to 1. A long-lived TAS object also provides a `Reset()` operation, which resets the bit. Only the process whose `TAS()` call returns 0 is allowed to subsequently call `Reset()`.

Chapter 6 presents two transformations of any one-time TAS object implemented from m registers into a long-lived one. The resulting long-lived TAS object of the first transformation uses $O(m+n)$ registers, `Reset()` takes $O(m)$ steps, and `TAS()` has asymptotically the same step complexity as in the original one-time object. The second transformation improves the step complexity of `Reset()` to $O(1)$, but the space complexity increases to $\Theta(mn)$. For example, applying the first transformation to the currently most space-efficient one-time TAS implementation (Giakkoupis, Helmi, Higham and Woelfel, 2015), we obtain a long-lived TAS object, where `Reset()` calls require $O(\log n)$ steps in the worst case, and `TAS()` calls have $O(\log^* n)$[1] expected step complexity against the oblivious adversary. The resulting object uses $O(n)$ registers, which is *optimal* due to the lower bound on the space requirement of mutual exclusion (Burns and Lynch, 1993). The second transformation yields a long-lived TAS that uses $O(n \log n)$ registers, `Reset()` runs in $O(1)$ steps in the worst case, and `TAS()` has again $O(\log^* n)$ expected step complexity against the oblivious adversary.

Moreover, some additional fine-tuning applied to a specific one-time TAS implementation by Giakkoupis and Woelfel (2012) and Alistarh and Aspnes (2011), allows us to create a long-lived TAS object with asymptotically optimal space, and expected $O(\log\log n)$ step complexity for both `TAS()` and `Reset()` against the oblivious adversary. Prior to these, no long-lived TAS

[1] $\log^* n$ denotes the *iterated logarithm* of n, i.e., the number of times $\log_2$ must be applied iteratively until the resulting value is at most 1.

implementation from $O(n)$ registers was known, where all operations have sub-linear expected step complexity.

An important part of all these implementations is an internal memory management technique to bound the space by recycling "used" objects when they are no longer in use. Although similar, each of them is specifically and differently tailored towards each of these applications. We exploit the main idea of our recycling technique, and combine it with other ideas to offer solutions, in the form of new primitives, to the following common problems that arise in algorithms with bounded space.

ABA Detection. The first one is the *ABA problem* which is a commonly encountered, challenging problem in the design of space bounded algorithms: Even though a process retrieves the same value twice in a row from a shared memory object, it is still possible that the value of the object has changed multiple times between these two retrievals. Algorithms using registers or compare-and-swap (CAS) objects seem to be especially susceptible. A CAS object stores a value C, and provides two operations, `Read()` and `CAS()`. The `Read()` operation returns the value of C. The `CAS(`x,y`)` operation changes the value of C to y and returns true if $C = x$ immediately prior to this operation. Otherwise, it leaves the object unchanged and returns false. A `CAS()` call is successful if and only if it returns true. Often, CAS objects are used in the following way: First, a process p reads the value x stored in the CAS object, then it performs some computation, and finally it tries to propagate the result of the computation y by performing a `CAS(`x,y`)`. The idea is that if another process has already updated the value of the object, p's `CAS()` should fail. However, if multiple successful `CAS()` operations have occurred and the value of the object has changed back to x, p's `CAS()` might still succeed, possibly yielding inconsistencies.

Tagging, as introduced by IBM (1983), provides a simple solution for the ABA problem. This involves augmenting an object with a tag, which is sometimes called a sequence number, that gets incremented with every change in the value of the object. If tags never repeat, this technique avoids the ABA problem. Therefore, theoretically, an infinite number of tags, and consequently,

base objects of unbounded size, are required. One may argue that, in practice, for reasonably large base objects, a system will never run out of tags. However, this is unrealistic in cases where the tag has to be stored together with other information in the same object. Alternatively, in some cases, it is possible to store the tag in a separate object (Jayanti and Petrovic, 2003; Michael, 2004c). However, this requires technically difficult algorithms and correctness proofs for each such application. Moreover, it is often necessary to use the entire object space for data, and then no space remains for tags. Some architectures like the IBM System/370 (IBM, 1983) have a double-width CAS primitive, which allows one of two words to be used for storing tags. But this primitive is not supported by most mainstream architectures (Michael, 2004a).

In contrast, load-link/store-conditional (LL/SC) objects do not suffer from the ABA problem. This type provides two operations, `LL()` and `SC()`. The `LL()` operation returns the current value of the object. The `SC(`x`)` operation may either fail, not change anything, and return `false`, or succeed, write the value x to the object, and return `true`. An `SC(`x`)` operation by process p succeeds if and only if no other `SC()` operation succeeded since p's last `LL()`. An extended specification also allows for a `VL()` (Verify-Link) operation, which does not change the state of the object; it returns `false` if a successful `SC()` has been performed since the calling process' last `LL()`, and `true` otherwise. LL/SC (or LL/SC/VL) objects can in almost all cases replace CAS objects in algorithms, and are an effective way of avoiding the ABA problem. Unfortunately, existing multiprocessor systems only provide weak versions of LL/SC that restrict programmers severely in how they can use the objects (Moir, 1997), and hence they "offer little or no help with preventing the ABA problem" (Michael, 2004a).

To study this problem more systematically, and investigate whether an LL/SC is necessary for avoiding the ABA problem, Chapter 5 introduces a new primitive that encapsulates the essentials to detect ABAs. This primitive is called an ABA-detecting register . It is similar to a normal read/write register, except that it allows a process to detect an ABA with a read operation. More specifically, this object provides two operations, `DRead()` and `DWrite()`. A process can write

to this object with a `DWrite()` operation. A `DRead()` by a process p returns the current value of the object, as well as a bit that indicates whether a process executed a `DWrite()` call since p's last `DRead()`. Although an ABA-detecting register can easily be implemented from a single unbounded register, Aghazadeh and Woelfel (2015) show that an obstruction-free implementation of even a single-bit ABA-detecting register requires at least $n-1$ bounded registers. In Chapter 5, we provide a wait-free implementation of an ABA-detecting register from $n+1$ bounded registers, with constant step complexity for both `DRead()` and `DWrite()` operations. This primitive can significantly help algorithm designers to focus on the main goal rather than being concerned about the ABA problem. In fact, this primitive is already used in our other constructions in this dissertation (in Chapters 6 and 7).

Independent of this thesis, Aghazadeh and Woelfel (2015) also show that any implementation of an ABA-detecting register, as well as an LL/SC object, from bounded CAS objects and registers, has an $n-1$ time-space tradeoff lower bound. This lower bound for LL/SC is asymptotically tight for implementations with constant step complexity, as it matches the known upper bound by Jayanti and Petrovic (2003). In Chapter 5, we show that a single bounded CAS object is enough to implement an LL/SC/VL object with $O(n)$ step complexity, which proves that this tradeoff lower bound is also asymptotically tight for implementations with constant space complexity.

We also show that a single LL/SC/VL object is enough to implement an ABA-detecting register, with constant step complexity. Therefore, an ABA-detecting register can also be implemented with a single CAS object in $O(n)$ steps, or with a CAS object and $O(n)$ registers in constant steps (using the implementation of LL/SC/VL object by Jayanti and Petrovic (2003)). This also shows that the time-space tradeoff lower bound by Aghazadeh and Woelfel (2015) for ABA-detecting registers is asymptotically tight for implementations with $O(1)$ and with $O(n)$ step complexities.

Bounded Tagging. In Chapter 7, we turn our attention to a more general definition of tagging, where registers or other shared objects get augmented with additional values, called tags.

Although one specific application of this technique could be to avoid the ABA problem, as it was initially suggested by IBM (1983), this technique could be used in many different settings.

For instance, in many algorithms, processes proceed in rounds. When a process writes to shared objects, it can augment its data value with a tag, which consists of the process's current round number and possibly process ID. Other processes can then use the tags read from shared objects to distinguish whether the corresponding data values were written by a process in the same or in different rounds. Each time a process starts a new round, it needs to find a new unique tag that it can use throughout its round as an identifier of that round. In some applications, it is important that tags are ordered, e.g. later rounds should have larger tags than earlier ones (Afek, Gafni, Tromp and Vitányi, 1992; Lamport, 1974; Riany, Shavit and Touitou, 2001; Vitányi and Awerbuch, 1986), while in others, the only requirement is that tags are unique (Aghazadeh and Woelfel, 2014; Attiya and Rachman, 1998; Giakkoupis and Woelfel, 2014).

A standard technique for tagging could be as follows: To generate a new tag a process p increments a local variable c, and then uses (p,c) as the tag. But in many algorithms this leads to an unbounded number of tags, and thus shared base objects need to be able to store values of unbounded size.

The problem of bounding tags is a special case of the bounded timestamp problem (Israeli and Li, 1987), in which processes can repeatedly call an operation which returns a timestamp, in such a way that the value of timestamps indicate the order they are received. While for tagging only uniqueness is essential, a temporal order relation must be satisfied for timestamps. In all known algorithms (for instance implementations by Dwork, Herlihy and Waarts (1993), Dolev and Shavit (1997), Dwork and Waarts (1992), Haldar and Vitányi (2002), and Israeli and Pinhasov (1992)), the operations to maintain bounded timestamps for n processes have step complexity at least $\Omega(n)$. Moreover, any timestamp system requires 2^n timestamps (Israeli and Li, 1987), and thus they must be stored in objects of size at least n bits. This limits the number of processes that can participate in algorithms relying on timestamp systems, especially if the timestamps need to

be stored in the same object as the data they are augmenting.

Several applications that use some specific variants of tagging rely on ad-hoc techniques to recycle tags with only constant step complexity overhead. Examples are implementations of LL/SC objects from CAS objects and registers (Anderson and Moir, 1995; Jayanti and Petrovic, 2003), FIFO queues from CAS objects and registers (Tsigas and Zhang, 2001), our transformations that augment objects with concurrent `Write()` or `Reset()` operations presented in Chapters 4 and 6, and our ABA-detecting registers provided in Chapter 5. This suggests that tagging could be easier to achieve than bounded timestamps.

To investigate this, we propose in Chapter 7 new primitives that capture the functionality we expect from tagging. Those primitives maintain a bounded pool of tags and an array A. Processes can obtain free tags from this pool. These tags can be stored alongside the data values in entries of array A. Any process that reads a tag from an entry of A, automatically protects it from being freed, until the process explicitly unprotects this tag. A process can store a tag that it read from an entry of this array or received from the pool, in another entry of the taggable array. When the tag is no longer needed, a process can release it, so that once no process has this tag protected, it goes back to the pool.

The operations that processes use to store and retrieve (data,tag) pairs from these primitives depend on the type of the array entries. In particular, we specify two variants of our types. We introduce the type taggable register array (TRA), in which A is an array of registers, and the type taggable LL/SC array (TLSA), in which A is an array of LL/SC objects.

In Chapter 7, we formally specify these types, and present an implementation of TRA from registers, and an implementation of TLSA from LL/SC objects and registers. Each of those implementations is wait-free, all operations have constant step complexity, shared base objects have bounded size (typically it is logarithmic in the number of processes), and the number of base objects used is bounded (typically polynomially in the number of processes).

These primitives can be used in many different applications, and could have a significant

impact on shared memory algorithm design and complexity analysis. For instance, they can be employed to enable memory management for a large class of algorithms called *single compare-and-swap universal (SCU)* (Alistarh, Censor-Hillel and Shavit, 2016). These primitives can also take over the memory management of some concurrent data structures. However, to completely enable memory management for most linked data structures, we need an extended version of our primitives, as explained in details in Chapter 7. The resulting primitive can be used in the same way for those data structures as other techniques like Hazard pointers by Michael (2004b) or Pass-the-Buck by Herlihy, Luchangco and Moir (2002).

Chapter 2 overviews the model, definitions, and assumptions of this work, and Chapter 3 presents the related work. In Chapter 8, we provide a summary of results and techniques proposed in this work, and outline some future directions that can be followed.

Contributions. In summary, the main contributions of this dissertation are as follows:

- Chapter 4: A general transformation to augment any object with a `Write()`/`Reset()` operation, provided that it is possible to write to/reset the original object sequentially (Aghazadeh, Golab and Woelfel, 2013, 2014).

- Chapter 5: Specification and an implementation (with asymptotically optimal space and step complexities) of a new primitive called ABA-detecting register, which provides a simple solution to the ABA problem (Aghazadeh and Woelfel, 2015).

- Chapter 6: Efficient transformations of any one-time test-and-set object to a long-lived one, as well as an optimal space implementation of a long-lived test-and-set object (Aghazadeh and Woelfel, 2014).

- Chapter 7: Specifications and efficient implementations of new primitives called taggable LL/SC array and taggable register array as general solutions to the bounded tagging problem (Aghazadeh and Woelfel, 2016).

Chapter 2

Model

2.1 General Definitions

This chapter provides the necessary definitions and assumptions that are used in this dissertation. The definitions in this chapter are based on the work by Attiya and Welch (2004), Golab, Higham and Woelfel (2011a,b), Herlihy and Shavit (2008), Herlihy and Wing (1990), and Woelfel (2017), even though in some cases we slightly deviate from their original definitions to accommodate our proofs.

2.1.1 Shared Memory Model and Shared Objects

We consider an asynchronous shared memory model with n processes with distinct IDs in $\mathcal{P} = \{0,\ldots,n-1\}$. Processes communicate through shared data structures, called *objects*. Processes are sequential, i.e. each process executes a sequence of operations on shared objects, repeatedly invoking an operation and then receiving the associated response.

An object has a *name* and a *type* that specifies the behaviour of the object. A type $\mathcal{T}$ is a quintuple $(\mathcal{S},s_0,\mathcal{O},\mathcal{R},\delta)$, where $\mathcal{S}$ is a set of *states*, $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{O}$ is a set of operations, and $\mathcal{R}$ is the set of responses. The set $\mathcal{R}$ contains a special value λ, which indicates that the operation does not return anything. The transition function $\delta : \mathcal{S} \times \mathcal{O} \rightarrow 2^{\mathcal{S}\times\mathcal{R}}$ defines the behaviour of the object of type $\mathcal{T}$, when processes execute operations sequentially. In this thesis, we restrict ourselves to deterministic types, in which $\delta : \mathcal{S} \times \mathcal{O} \rightarrow \mathcal{S} \times \mathcal{R}$. For a deterministic type $\mathcal{T} = (\mathcal{S},s_0,\mathcal{O},\mathcal{R},\delta)$, if a process executes an operation $Op \in \mathcal{O}$, when the object is in state $s \in \mathcal{S}$, then the operation returns the value $r \in \mathcal{R}$ and the state of the object changes to $s' \in \mathcal{S}$, if and only if $\delta(s,Op) = (s',r)$.

Most often, a more descriptive approach is used to describe the type of an object: In this

approach plain language is used, as opposed to an automaton description, to specify the value that the object can store, as well as the operations it provides, and how these operations affect the stored value in a sequential execution.

We call the shared objects that are provided by the system *base objects*. Some shared objects are implemented from other base objects. Section 2.1.3 outlines what it means for an implementation to be correct. In the following, we introduce the shared objects that are used or implemented in this dissertation.

Read-Write Registers. A read-write register, or commonly called an atomic register, stores a value (usually of size one-word) and provides two operations: A `Read()` operation returns the current value of the object, and a `Write(`x`)` changes the value of the object to x. A k-word register stores a value of size k-words, and a `Read()` operation returns the k words that are currently stored in the object, and `Write(`x_1`,...,`x_k`)` takes k 1-word values and updates the value of the object correspondingly. It is historically interesting to specify how many readers can read from, and how many writers can write to a register. We distinguish between single-writer single-reader (SWSR), single-writer multi-reader (SWMR), and multi-writer multi-reader (MWMR) registers. In this work, we assume MWMR atomic registers are provided by the system.

Compare-and-Swap. A compare-and-swap (CAS) object stores a value, and supports two operations: A `Read()` returns the value of the object, and a `CAS(`x`,`y`)` operation succeeds and changes the value of the object to y if the current value of the object is x and returns true, otherwise it fails and leaves the object unchanged and returns false. According to some specifications, a `CAS()` always returns the prior value of the object. In this thesis, we assume `CAS()` operations return only boolean values.

Load-Link/Store-Conditional. A load-linked/store-conditional (LL/SC) object stores a value and provides two operations `LL()` and `SC()`. An `LL()` initiates a link to the object and returns the current value of it. An `SC(`x`)` may either succeed and write value x to the object, or fail and

not change anything. A successful `SC()` call invalidates all the links processes hold to the object. An `SC(`x`)` operation call succeeds if and only if the calling process has a valid link the object. In other words, an `SC(`x`)` operation call by process p succeeds if and only if p previously executed an `LL()` call and since then no other successful `SC()` call was performed. A boolean return value of a `SC()` operation indicates if it succeeded. We assume initially no process has a valid link to the object, and so an `SC()` call without an earlier `LL()` call by the same process fails.

A variant of this object is called load-link/store-conditional/validate (LL/SC/VL). This variant supports one additional operation: `VL()` returns true if and only if the calling process holds a valid link to the object.

Test-and-Set. A test-and-set (TAS) object stores a bit, which is initially 0. The one-time version allows only one operation `TAS()`, which sets a bit to 1, and returns the previous value of that bit. A long-lived TAS object also provides a `Reset()` operation, which resets the bit to 0 (and has no return value). We say a process *wins* a `TAS()` if that operation call returns 0, otherwise it *loses*. Only the process that *wins* a `TAS()` operation call is allowed to subsequently reset it (Afek, Gafni, Tromp and Vitányi, 1992).

Fetch-and-Add. A fetch-and-add (FAA) object stores a value, which is initially 0. This object supports one operation `FAA(`x`)`, which atomically returns the current value of the object and adds x to it.

2.1.2 Schedule, Transcript and History

Processes execute shared memory operations on shared objects in the order determined by their program. In randomized algorithms, processes can make random decisions using (private) coin flips as a source of randomness. A *schedule* determines the order in which processes take turns in executing their operations. In the case of randomized algorithms, the schedule is determined by an *adversary*, in response to random choices made by processes. The *strong adaptive* adversary makes scheduling decisions based on the entire past execution history including the results of

coin-flips. An *oblivious* adversary has to determine the entire schedule independently of random decisions.

We model an execution of a concurrent shared memory system with a *transcript*, which is a finite sequence of invocation and response events on base objects as well as implemented objects. Let Λ be a transcript. A subsequence of the events of Λ is called a *subtranscript* of Λ. Each invocation event in Λ has a distinct ID, and specifies which process is executing what operation on which object. Each response event in Λ stores the ID of an invocation event in Λ as well as an output of the operation call associated with that invocation. A response event *matches* an invocation event if it stores the ID of that invocation event. An *operation call* in Λ is either a pair consisting of an invocation and a matching response event in Λ, or just an invocation event if that invocation does not have a matching response event in Λ. An operation call is *atomic* if its invocation event is immediately followed by the matching response event in Λ. We say a process executes a shared memory *step*, when it makes an atomic operation call.

Consider a transcript Λ by processes in $\mathcal{P}$ that contains invocation and response events of operation calls on objects $B_1, \ldots, B_k$, for some integer $k \geq 1$. We let $\Lambda|P$ (Λ at P) denote the subsequence of all events in Λ that are by processes in $P \subseteq \mathcal{P}$. We similarly define subtranscript $\Lambda|B$ to be the subsequence of all events in Λ that are on objects of some set $B \subseteq \{B_1, \ldots, B_k\}$. Let O be a set of operation calls in Λ. Then subtranscript $\Lambda|O$ is the subsequence of all invocation and response events of operation calls in O in Λ. For a singleton set $\{s\}$, we write $\Lambda|s$ instead of $\Lambda|\{s\}$.

For any $p \in \mathcal{P}$, subtranscript $\Lambda|p$ is *well-formed* if each response event in $\Lambda|p$ matches exactly one earlier invocation event, for each invocation event inv in $\Lambda|p$ there is at most one matching response event rsp, and one of the following is true:

- inv is immediately followed by rsp,
- inv is followed by an invocation event inv', and if rsp appears in $\Lambda|p$, then a response rsp' that matches inv' appears before rsp, or

- inv is the last event in $\Lambda|p$.

Transcript Λ is well-formed if $\Lambda|p$ is well-formed for all $p \in \mathcal{P}$. From now on, we assume all transcripts are well-formed.

An operation call Op is *pending* in Λ if its invocation event has no matching response event in Λ. Otherwise, it is complete in Λ. If all operation calls in Λ are complete, then Λ is a *complete* transcript. For any transcript Λ, a *completion* of Λ is a (well-formed) transcript constructed from Λ such that each pending operation is either removed, or a matching response is appended.

An operation call Op_1 *happens before* (or *precedes*) an operation call Op_2 in Λ if the response event of Op_1 appears before the invocation event of Op_2 in Λ. Let $\stackrel{\Lambda}{\rightarrow}$ be the relation over all operation calls in Λ, such that $Op_1 \stackrel{\Lambda}{\rightarrow} Op_2$ if and only if Op_1 happens before Op_2 in Λ. Then for any (well formed) transcript Λ, the relation $\stackrel{\Lambda}{\rightarrow}$ is an irreflexive partial order, called the *happens before order*. Operation calls Op_1 and Op_2 are *concurrent* (or they *overlap*) if neither of them happens before the other one in Λ. An operation call is executed in *isolation* if it does not overlap any other operation call. A transcript is *sequential* if no two operation calls are concurrent.

A well-formed *history* is a well-formed transcript Λ, where in $\Lambda|p$, for all $p \in \mathcal{P}$, each invocation event is either the last event, or is immediately followed by a matching response event. A sequential history is one that starts with an invocation event, and every invocation event except for possibly the last one is immediately followed by a matching response event. A *sequential specification* for an object is a set of sequential histories for the object. A sequential history H on objects $B_1,\ldots,B_k$, for some $k \geq 1$, is *valid* if subhistory $H|\{B_i\}$ belongs to the sequential specification for object B_i, for all $i \in \{1,\ldots,k\}$.

A transcript Λ has an *interpreted* history $\Gamma(\Lambda)$, defined as follows: As long as Λ contains an invocation event inv by some process p, remove any event e from Λ such that e appears after inv in $\Lambda|p$, and

(a) there is no matching response for inv, or

(b) there is a matching response for inv and e appears before that response event.

2.1.3 Safety

Informally, the safety criteria ensures that an implementation behaves as it is specified by its type. The safety property that we work with is *linearizability* introduced by Herlihy and Wing (1990). Let H be an arbitrary (well-formed) history. A sequential history S is a *linearization* of H if there is a completion H' of H, such that

(a) S contains the same events as H',

(b) S is valid, and

(c) the happens before order of operation calls in H is preserved in S, that is if $Op_1 \xrightarrow{H} Op_2$ then $Op_1 \xrightarrow{S} Op_2$.

History H is *linearizable* if it has a linearization.

We can prove that a complete history H is linearizable, using *linearization points*, as described below. Informally, a linearization point of an operation call is the point at which the operation call seems to take effect. More specifically, we map each event e of history H to a non-negative real number $\mathtt{pt}(e)$, such that it satisfies the following *monotonicity requirement*:

$$\mathtt{pt}(e) < \mathtt{pt}(e'), \text{ for any two events } e \text{ and } e' \text{ in } H, \text{ where } e \text{ is immediately followed by } e'. \quad (2.1)$$

Any mapping satisfying the monotonicity requirement can be used. For example, if e_i is the i-th event in H, we can use $\mathtt{pt}(e_i) = i$. We think of $\mathtt{pt}(e)$ as the point in time at which event e occurs. History H is linearizable if for each operation call Op, with invocation event inv and response event rsp, there exists a point $lin(Op) \in [\mathtt{pt}(inv), \mathtt{pt}(rsp)]$, such that the sequence of operation calls in H ordered by their $lin()$ points is a valid sequential history. In that case, $lin(Op)$ is called the chosen linearization point of operation call Op. Figure 2.1 shows an example execution of an implemented shared queue object, and one possible assignment of linearization points. The resulting sequential history obtained by ordering operations by their assigned linearization points is valid, and so this execution is linearizable.

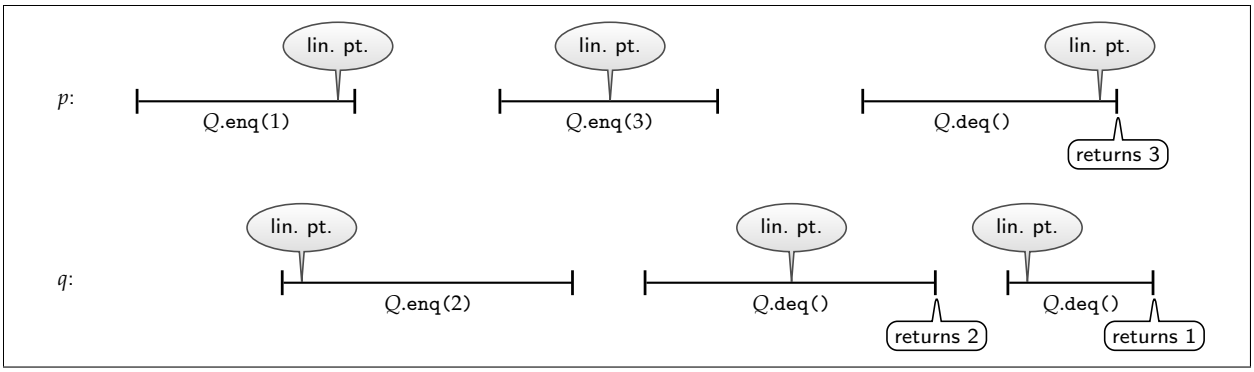

Figure 2.1: An example execution of an implemented queue with a possible linearization points assignment

An implemented object X is linearizable if for every transcript Λ obtained from executing operations on X, $\Gamma(\Lambda)$ is linearizable.

Linearizability is *compositional*:

Theorem 2.1. *(Herlihy and Wing, 1990) A history H is linearizable if and only if $H|x$ is linearizable, for every object x.*

The following is a well known property of linearizaility which follows from the definition of linearization points and Theorem 2.1.

Theorem 2.2. *If in a linearizable implementation, an atomic base object is replaced with a linearizable one, then the resulting implementation is also linearizable.*

These properties of linearizability make it the most commonly used correctness condition compared to other existing correctness conditions, for instance sequential consistency (Lamport, 1979), which does not support composition. Linearizability allows programmers to design their deterministic algorithms assuming atomic base objects are available, and later replace those base objects with their linearizable implementations without changing the safety property of their algorithm. It is important to mention that this does not extend to randomized algorithms (Golab, Higham and Woelfel, 2011a,b).

2.1.4 Progress Conditions

Algorithm designers are interested in algorithms that provide some progress guarantees. The strongest progress condition is *wait-freedom*. In a wait-free implementation, each process finishes its operation call after executing a finite number of steps. This is an attractive progress condition, because it guarantees that each process that is taking steps progresses.

An implementation of an object is *lock-free* if at least one process is guaranteed to finish its operation call provided that there exists processes that take sufficiently many steps. In such an implementation, some processes could starve. In an *obstruction-free* implementation, if at any point during an operation call a process continues in isolation, then the process finishes its operation call in a finite number of steps. *Deadlock-freedom* does not prevent processes from blocking some other processes from progressing. A deadlock-free implementation guarantees that *at least* one process finishes its operation call if *all* other processes that are concurrently accessing the object take sufficiently many steps. In a *starvation-free* implementation, *each* process has to eventually finish its operation call if *all* other processes that are concurrently accessing the object take sufficiently many steps. A randomized implementation is *randomized wait-free* if for each operation call by each process, the expected number of steps for the process to finish its operation call is finite. A nondeterministic implementation satisfies *nondeterministic solo-termination* if at any point during an operation call, it is possible for the calling process to finish its operation call in isolation. For deterministic implementations, nondeterministic solo-termination is the same as obstruction-freedom.

2.1.5 Space and Step Complexity

For an implemented object, the step complexity of an operation is the maximum number of shared memory steps a process takes to finish that operation.

For randomized algorithms scheduled by some adversary A, the expected step complexity of an operation is the worst case over all possible schedules generated by A of the expectation over

all possible random coin flips of the step complexity of that operation. The space complexity of an implementation is the maximum number of base objects required for that implementation.

2.2 Definitions Specific to Our Linearizability Proofs

To prove that an implemented object X is linearizable, we show for each transcript Λ obtained from executing operations on X, that $\Gamma(\Lambda)$ is linearizable. To that end, we use the following approach to define a mapping, $\mathtt{pt}$, of events of $\Gamma(\Lambda)$ to non-negative real numbers, which will then allow us to define linearization points.

For our linearizability proofs, we will only encounter transcripts where the innermost operation calls are either atomic or linearizable. If they are linearizable, we can replace each innermost non-atomic operation call Op with an atomic one, that is executed at the linearization point of Op. Therefore, we assume every innermost operation call in Λ is atomic.

Suppose $\Lambda = (e_1, e_2, \ldots)$. To define the mapping $\mathtt{pt}$ for $\Gamma(\Lambda)$, we first define another mapping $\mathtt{pt}'$ for Λ by letting $\mathtt{pt}'(e_i) = i$. Next, for each operation call Op in Λ, we define a point $t_{Op@inv}$ that corresponds to the invocation event of Op, and if the response event of Op is in Λ, a point $t_{Op@rsp}$ that corresponds to the response event of this operation call, as follows. If Op is an atomic operation call with invocation event inv, then we let $t_{Op@inv} = t_{Op@rsp} = \mathtt{pt}'(inv)$. If Op is not atomic, then we first recursively determine $t_{Op'@inv}$, where Op' is the operation call with the earliest invocation event following the invocation event of Op in $\Lambda|p$, and then we let $t_{Op@inv} = t_{Op'@inv}$. If Op also has a response event in Λ, then we first recursively determine $t_{Op'@rsp}$, where Op' is the operation call with the latest response event preceding the response event of Op in $\Lambda|p$, and then we let $t_{Op@rsp} = t_{Op'@rsp}$. If Op does not respond in Λ, then we let $t_{Op@rsp} = \infty$. Therefore, for any completed operation call Op in Λ with invocation event inv and response event rsp, we have

$$\mathtt{pt}'(inv) \leq t_{Op@inv} \leq t_{Op@rsp} < \mathtt{pt}'(rsp). \tag{2.2}$$

For $H = \Gamma(\Lambda)$, and any event e in H, let $\mathtt{pt}(e) = \mathtt{pt}'(e)$. The monotonicity requirement

of (2.1) is satisfied for $\mathtt{pt}$, because it is satisfied for $\mathtt{pt}'$. Thus by (2.2), to prove linearizability of an implemented object X, it suffices to find a point $lin(Op) \in [t_{Op@inv}, t_{Op@rsp}] \subseteq [\mathtt{pt}(inv), \mathtt{pt}(rsp)]$, and prove that ordering operation calls Op of $\Gamma(\Lambda)$ based on their values $lin(Op)$ creates a valid sequential history.

In our linearizability proofs, we assume that at each point in time when a process executes an atomic operation call on a shared object, it also executes all its following local steps (operations on local variables) instantaneously, until it is poised to execute its next atomic operation call on a shared object. We can make this assumption, because any intervening steps from other processes cannot influence these local steps. We say an operation call Op by some process p *completes* in some interval $[t, t']$ if $[t_{Op@inv}, t_{Op@rsp}] \subseteq [t, t']$.

We allow the following in our proofs, for any operation call Op by some process p, either executed on an object or as a helper operation. Let ℓ be a line of code executed during Op. If p executes exactly one atomic operation call Op' on a shared object in line ℓ, then we let $t_{Op@\ell} = t_{Op'@inv}$. If p executes only operations on local variables in line ℓ, we let $t_{Op@\ell} = t_{Op@inv}$ if ℓ is the first line in Op, and otherwise $t_{Op@\ell} = t_{Op@\ell-1}$. If p executes more than one atomic operation call on a shared object in line ℓ of Op, then $t_{Op@\ell}$ is not defined. We say that p executes line ℓ of operation call Op at point $t_{Op@\ell}$ if $t_{Op@\ell}$ is defined. Finally, we let $t_{Op@\ell} = \infty$ if the execution of Op does not reach line ℓ.

In our pseudocodes, by convention, we let shared variables and public operations of objects start with an upper-case character, and local variables and internal operations start with a lower-case character. We declare those local variables that have global scope in the code, but not the temporary local variables.

Chapter 3

Related Research

3.1 Augmenting Objects with Write() Operations

The problem of augmenting a type with a `Reset()` or a `Write()` operation discussed in Chapter 4, has, to the best of our knowledge, not been studied in general, but only for some specific types. Alistarh and Aspnes (2011), Afek, Gafni, Tromp and Vitányi (1992), and Hoepman (1999) propose implementations of long-lived test-and-set objects from one-time ones, as we describe in more details later. Golab, Hadzilacos, Hendler and Woelfel (2012) consider writable implementations of compare-and-swap (CAS) objects from atomic registers. Their approach augments a non-writable CAS implementation with a `Write()` operation using pointer swinging and dynamically allocated non-writable CAS base objects. The corresponding memory management technique, discussed in Golab's doctoral thesis, relies on a critical section (Golab, 2010), and hence cannot be used in lock-free structures. Jayanti (1998) shows how to implement a CAS object that supports `CAS()`, `Read()`, and `Write()` operations, from LL/SC/VL objects. This implementation has constant step and space complexity.

Our results regarding multi-word atomic registers of Section 4.1.1 are related to wait-free register constructions. The construction of single-writer multi-reader (SWMR) k-word registers by Peterson (1983) requires $\Theta(kn)$ SWMR single-word registers and provides $\Theta(kn)$ worst-case step complexity. Using stronger primitives than atomic registers, Larsson, Gidenstam, Ha, Papatriantafilou and Tsigas (2008) propose an SWMR k-word atomic register implementation with $\Theta(kn)$ space complexity and $\Theta(k+n)$ worst-case step complexity. In comparison, our construction (see Corollary 4.4) is multi-reader multi-writer (MRMW), requires $\Theta(kn^2)$ MRMW single-word atomic registers, and has $\Theta(k)$ worst-case step complexity. This is the first implementation of k-word registers with optimal $O(k)$ worst-case step complexity, and bounded space.

Zhu and Ellen (2015) build upon our multi-word register implementation to achieve an efficient snapshot implementation from single-word registers. In a later work, Chen and Wei (2016) show how to implement a single-writer k-bit register from $O(nk/\ell)$ registers, each of size ℓ bits, with $O(k/\ell)$ step complexity. They show that the same step and space complexity can be achieved for $\ell = \Omega(\log n)$ by a simple modification of our implementation.

3.2 Test-and-Set

Test-and-set (TAS) objects have many algorithmic applications, for example in mutual exclusion and renaming algorithms (Alistarh, Aspnes, Censor-Hillel, Gilbert and Zadimoghaddam, 2011a; Alistarh, Aspnes, Gilbert and Guerraoui, 2011b; Alistarh, Attiya, Gilbert, Giurgiu and Guerraoui, 2010; Buhrman, Panconesi, Silvestri and Vitányi, 2006; Eberly, Higham and Warpechowska-Gruca, 1998; Kruskal, Rudolph and Snir, 1988; Panconesi, Papatriantafilou, Tsigas and Vitányi, 1998).

One-time test-and-set objects and registers can be used to implement two-process consensus and vice versa, so it follows that there is no deterministic wait-free linearizable implementation of TAS objects from registers (Herlihy, 1991; Loui and Abu-Amara, 1987). A randomized two-process long-lived TAS implementation with constant space and constant expected step complexity, is given by Tromp and Vitányi (2002), and is used in almost all n-process TAS implementations. The famous randomized one-time n-process TAS implementation by Afek, Gafni, Tromp and Vitányi (1992) is based on a tournament tree consisting of two-process TAS objects. This implementation uses $O(n)$ registers, and the `TAS()` method takes $O(\log n)$ steps in expectation against a strong adaptive adversary. Their long-lived variant provides a constant time `Reset()` method but increases the expected step complexity of the `TAS()` method to $O(n)$, and requires registers of size at least $\Omega(n)$. Alistarh, Attiya, Gilbert, Giurgiu and Guerraoui (2010) proposed a one-time variant of that tournament tree in which the expected step complexity of the `TAS()` method is logarithmic in the contention against the strong adaptive adversary. Their

construction needs $O(n^3)$ registers.

During the last couple of years, a series of papers focused on improving the time- and space-complexities of one-time TAS implementations in weaker adversary models (Alistarh and Aspnes, 2011; Giakkoupis, Helmi, Higham and Woelfel, 2013, 2015; Giakkoupis and Woelfel, 2012). In particular for weaker adversary models one-time TAS implementations with almost constant step complexity are devised. Alistarh and Aspnes (2011) propose a `TAS()` method that has $O(\log\log n)$ expected step complexity against the oblivious adversary, using $O(n^3)$ registers. Following this work, Giakkoupis and Woelfel (2012) reduce the space to $O(n)$ and provide an adaptive version of the algorithm with an expected step complexity of $O(\log\log k)$ against the oblivious adversary, where k is the contention. In fact, these double-logarithmic algorithms by Alistarh and Aspnes (2011) and Giakkoupis and Woelfel (2012) achieve the claimed step complexities even against the slightly stronger read-write oblivious adversary. Giakkoupis and Woelfel (2012) also present an adaptive one-time TAS algorithm that has $O(\log^* k)$ expected step complexity against the oblivious (and also a slightly stronger location-oblivious) adversary, using $O(n)$ registers. Giakkoupis, Helmi, Higham and Woelfel (2013) show how to reduce the space complexity of this object to $O(\sqrt{n})$ registers, while maintaining the expected step complexity of $O(\log^* n)$. Finally, the randomized implementation by Giakkoupis, Helmi, Higham and Woelfel (2015) hits the asymptotically optimal space complexity of $O(\log n)$ (Styer and Peterson, 1989), while `TAS()` calls require only $O(\log^* n)$ steps in expectation.

However, research on efficient implementations of long-lived TAS objects has trailed behind. In fact, prior to our work in Chapter 6, no long-lived construction was known, which achieves $O(n)$ space complexity, and at the same time provides `TAS()` and `Reset()` operations with sub-linear expected step complexity. The space complexity of $O(n)$ is asymptotically optimal as implied by the $\Omega(n)$ lower bound for mutual exclusion by Burns and Lynch (1993).

A long-lived implementation of a TAS object using registers was first presented by Afek, Gafni, Tromp and Vitányi (1992). Although their construction uses registers of unbounded size, they

suggest adopting a modified version of the Sequential Timestamps System (STSS) by Israeli and Li (1987) to bound the size of registers used in this implementation. Their long-lived implementation requires $\Theta(n)$ steps in expectation. Alistarh and Aspnes (2011) propose a simpler implementation of long-lived TAS by using an *infinite* array of one-time TAS objects. They briefly mention that the problem of infinite number of allocations can be resolved by "allowing the current winner to deallocate the current instance in the `Reset()` procedure, and adding version numbers to shared variables, so that processes executing a deallocated instance automatically return *loser*". But deallocating an instance of an implemented TAS object is no trivial task as one has to ensure that no process is poised to access the TAS object that is about to be de-allocated. It is unclear how to do this just with version numbers, and without having processes at least "announce" TAS objects that they are about to access, similarly to the Hazard Pointers technique (Michael, 2004b). Hoepman (1999) shows how to implement a long-lived TAS object from $O(n)$ registers and $n+1$ one-time TAS objects, provided those TAS objects can be reset in a sequential execution. The resulting `TAS()` method has, up to a constant additive term, the same step complexity as the `TAS()` method on the one-time TAS object, and the `Reset()` method requires n steps in addition to the number of steps it takes to sequentially reset the one-time TAS object.

3.3 Memory Management

The problem of memory management, which is of technical concern in our transformations, has been studied in the context of dynamic data structures, leading to a variety of solutions. The challenge is to decide when a shared object is no longer going to be accessed by processes, so that the memory location associated with it can be reused.

In *reference counting*, the number of pointers or references held to a given object is recorded, and an object may be reclaimed when the reference count reaches zero. There have been many proposed techniques (Detlefs, Martin, Moir and Steele Jr., 2002; Ellen, Lev, Luchangco and Moir, 2007; Lee, 2010; Sundell, 2005; Valois, 1995), but they all use stronger primitives than registers,

such as double-word compare-and-swap and fetch-and-add. A problem with reference counting is that reading one shared object by multiple processes can result in a potential contention for increasing the reference count of that object. Moreover, reference counting in general deals poorly with cyclic references, which occurs when two or more objects refer to each other.

The *Hazard Pointers* technique is targeted at general lock-free algorithms, and does not require atomic read-modify-write primitives (Michael, 2004d). Object reclamation is wait-free and requires expected amortized $O(1)$ steps, but dereferencing a pointer safely is only lock-free. Hazard pointers impose a fixed bound on the number of shared objects a given process can reference, and memory allocation must be provided by the system when adopting this technique. Each process uses a list of pointers to announce to other processes that it is accessing a particular shared object (to indicate a "hazard"). To access an object, a process first writes the object's address to a hazard pointer, and then checks whether the object remains safe to access. If not, the algorithm must give up or retry, and so as a result, this technique only achieves lock-freedom rather than the stronger wait-freedom. To decide whether an object is recyclable, a process scans the hazard pointers of all other processes, and frees those that do not appear as a hazard.

Concurrently and independently, Herlihy, Luchangco and Moir (2002) proposed a similar lock-free technique called Pass-the-Buck, which uses compare-and-swap objects. This technique is provided as a solution to the *repeat offender problem*, which captures a difficult race condition between a process reclaiming a shared object and another process attempting to dereference a pointer to that object. Herlihy, Luchangco, Martin and Moir (2005) show how this technique is used to recycle memory in a lock-free queue implementation by Michael and Scott (1996).

Epoch-based reclamation (EBR) and *quiescent-state-based reclamation* (QSBR) are similar in the sense that they reclaim memory once a grace period has passed. A grace period is an interval $[a,b]$, such that after point b, all objects removed before point a can be reclaimed (Hart, McKenney, Brown and Walpole, 2007). For EBR, the execution is divided into epochs, and with the invocation of each operation, the calling process reads and announces the current epoch. A

new epoch starts when all processes have announced the current one. With a new epoch, all objects removed in the last epoch can be reclaimed (Fraser, 2004). A process reaches a quiescent state when it does not hold any references to any objects. A grace period in QSBR is an interval in which all processes reach at least one quiescent state. QSBR is used in Linux to implement the Read-Copy-Update (RCU) method (McKenney and Slingwine, 1998). However, in both EBR and QSBR, a failed process can block all other processes from reaching a grace period, and so these techniques are inherently blocking and not fault-tolerant. Brown (2015) also proposes a fault-tolerant epoch bases technique called DEBRA+, which uses signals in order to enable processes to advance epochs. This technique also manages pools of memory blocks, and provides (de-)allocation methods, which are not supported in techniques like Hazard Pointers (Michael, 2004b). However, DEBRA+ uses lock-free linked lists to achieve that, and therefore is not wait-free, and its performance depends on the implementation of the list. The memory reclamation technique proposed by Balmau, Guerraoui, Herlihy and Zablotchi (2016) combines the idea of QSBR with Hazard Pointers, and uses timestamps to deal with slow or faulty processes.

Gidenstam, Papatriantafilou, Sundell and Tsigas (2009) proposed a lock-free memory reclamation method that combines aspects of reference counting and hazard pointers, and requires both, fetch-and-add and compare-and-swap, primitives. Braginsky, Kogan and Petrank (2013) use timestamps with a variant of Hazard Pointers to obtain memory management for linked lists. Alistarh, Leiserson, Matveev and Shavit (2017) propose a memory reclamation technique that uses modern operation system support for efficient signalling and copy-on-write to obtain a snapshot of the memory. An automatic lock-free memory reclamation technique is presented by Cohen and Petrank (2015) which works with data structures that are stored in normalized form.

Since in this work, we are dealing with bounding space, all our implementations depend on an efficient reclamation technique. Applying any of these techniques in our transformations either requires stronger primitives than atomic registers, or leads to larger step complexity. Hence, we introduce our own ad-hoc memory reclamation scheme in Chapter 4, called recycling technique,

which is internally used in all our implementations. This technique which was inspired by Hazard Pointers (Michael, 2004b), reclaims objects in constant step complexity, and uses only registers.

3.4 ABA Detection

The ABA problem was first reported by IBM (1983). The same report suggests a simple tagging solution: a tag value is attached to the data value that can potentially suffer from the ABA problem, and with every update of the data value, the tag is incremented. Theoretically, this scheme can only detect ABA if unbounded shared objects are available. However, it is commonly argued that this is not a practical concern with current hardware (Arbel-Raviv and Brown, 2017; Michael, 2004c). Although often solutions for dealing with ABAs are based on tagging (Hendler, Shavit and Yerushalmi, 2010; McKenney and Slingwine, 1998; Michael, 2002, 2004d; Michael and Scott, 1996; Prakash, Lee and Johnson, 1991; Shann, Huang and Chen, 2000; Stone, 1990), there are ad-hoc, application specific solutions to deal with ABAs (Tsigas and Zhang, 2001; Valois, 1995). Others suggest combining tagging and memory management techniques as alternatives (Hendler, Shavit and Yerushalmi, 2010; Ladan-Mozes and Shavit, 2008).

Although LL/SC/VL can be used to prevent the ABA problem, the existing hardware provides only a weaker version of this object, which restricts algorithm designers on how this object can be used (Doherty, Herlihy, Luchangco and Moir, 2004; Michael, 2004c; Moir, 1997). For instance, it is not possible to have nested or interleaving `LL()`/`SC()` calls (Michael, 2004a). To that end, researchers have been proposing to implement LL/SC/VL objects from CAS objects and registers. Such an implementation by Israeli and Rappoport (1994) uses shared objects of size $\Omega(n)$ bits. This limits the number of processes concurrently running an application which uses this implemented object. Implementations by Anderson and Moir (1995) and Moir (1997), have constant step and space complexities, and store a version number along with the object value to avoid ABAs. Jayanti and Petrovic (2003) argue that LL/SC/VL objects obtained from those implementations can only store small values (16-24 bits on a 64-bit architecture). Jayanti and

Petrovic (2003) also improve those implementations by achieving a 64 bit LL/SC/VL from one 64-bit CAS object and $O(n)$ registers, while maintaining constant step complexity. Michael (2004c) implements an arbitrary size LL/SC/VL from 64-bit CAS objects and registers, with constant amortized expected step complexity, and polynomial space complexity. These last two implementations use sequence numbers that are stored in separate memory locations, and they argue that it is "practically" impossible for the sequence numbers to repeat.

We introduce an implementation of a b-bit LL/SC/VL, for any $b \in \mathbb{N}$, from a single $(b+n)$-bit CAS object, with $O(n)$ step complexity. This is the first implementation that asymptotically matches the time-space tradeoff discussed earlier at the point with only one base object. Our implementation and the one by Jayanti and Petrovic (2003) show that the time-space tradeoff lower bound by Aghazadeh and Woelfel (2015) for implementing an LL/SC object from bounded CAS objects and registers is asymptotically tight for implementations with $O(1)$ and with $O(n)$ step complexities.

3.5 Tagging

Considering tags as memory addresses, our taggable objects from Chapter 7 can be directly used for memory reclamation in several algorithms, and allow certain operations to be wait-free, where other memory reclamation techniques guarantee only lock-freedom. The core reason is that retrieving the reference of a memory block and protecting it are separated into two distinct operations. This is different from our taggable object implementation: Retrieving a tag from a taggable register or taggable LL/SC object automatically protects the tag from being freed, and is achieved through a wait-free operation with constant step complexity.

Our abstraction works best if only a fixed number of nodes need to be protected, such as the root of a tree, or the top element of a stack. In the case of general linked data structures, where any node may have to be protected (e.g. a linked list), our taggable objects do not achieve better progress conditions than other memory reclamation techniques. (See also Section 7.2.3.)

Most memory reclamation techniques rely on the operating system to provide methods for allocation and deallocation of memory blocks. The time and space complexity of those allocation methods, or their progress guarantees, are not part of the analysis. Our taggable object implementation manages its own pool of tags, and does not rely on any external methods for memory allocation. Consequently, progress is not dependent on the progress of the system's memory allocation layer. A disadvantage of our implementations may be that a sufficiently large pool of shared objects needs to be preallocated. It is not hard to accommodate our implementations so that the pool of tags is dynamically resized, if the system provides methods for memory allocation and deallocation, but this is beyond the scope of this work.

As explained earlier, the problem of bounding tags is similar to bounded timestamping, however, timestamps must satisfy an additional temporal order relation. This functionality is costly: Any timestamp system requires at least 2^n timestamps (Israeli and Li, 1987), and thus objects of size at least n bits are required. Also in all known algorithms, the step complexity of operations to maintain bounded timestamps for n processes is at least $\Omega(n)$. Some solutions of bounded concurrent timestamp systems are based on precedence graph (Dolev and Shavit, 1997; Israeli and Pinhasov, 1992), while others (Dwork and Waarts, 1999; Haldar and Vitányi, 2002; Shikaripura and Kshemkalyani, 2002) are based on the *traceable use abstraction*.

The traceable use abstraction introduced by Dwork and Waarts (1992, 1999) provides a solution to bounded concurrent timestamp systems, and is semantically similar to our taggable register array primitive. The goal of this abstraction is to make the values processes write *traceable*. This means that each process, at any point in time, should be able to determine a nontrivial superset of the set of its values currently in use. In this abstraction processes can only write their values to single-writer registers.

The implementation provided by Dwork and Waarts (1999) for the traceable use abstraction employs a hint mechanism that is similar to the one we use in this work, but simpler. To synchronize between a reader and a writer, this implementation uses handshaking bits by Peterson

(1983) and Lamport (1986). Most operations in this implementation have at least linear step complexity.

Our hint mechanism (from Chapters 4 and 7) utilizes some different techniques instead of handshaking bits, to achieve constant step complexity for all operations. Our taggable abstraction achieves the same functionalities as the traceable use abstraction, and yet is more general and easier to be used. For instance, in the traceable use abstraction, processes can only write to single writer registers, while in our implementations, values can be shared in multi-writer registers. Moreover, in our taggable implementation, processes can hold references to the values that they read as long as there is a bound on the total number of values that all processes have a reference. But in the traceable use abstraction, as soon as a new read starts, all the old references are unsafe to be used.

Chapter 4

Writable Objects

Consider a type $\mathcal{T} = (\mathcal{S}, s_0, \mathcal{O}, \mathcal{R}, \delta)$ corresponding to a shared object. We call $\mathcal{T}$ *writable* if there is a `Write(s)` operation in $\mathcal{O}$ that changes the state of the object to state $s \in \mathcal{S}$, regardless of its current state. A `Write()` operation returns nothing.

An implementation of a writable type is *sequentially writable* if every history on that object is linearizable, provided that no `Write()` operation call overlaps any other operation call (see Figure 4.1). We will denote a `Write()` operation of a sequentially writable object $\mathcal{SW}$ by `SWrite()` to distinguish it from a `Write()` operation of a linearizable object $\mathcal{WT}$ of the same type, and to emphasize that $\mathcal{SW}$.`SWrite()` operation calls are not allowed to overlap with any other operation call on $\mathcal{SW}$.

Similarly, we say a type $\mathcal{T} = (\mathcal{S}, s_0, \mathcal{O}, \mathcal{R}, \delta)$ is *resettable* if $\mathcal{O}$ includes a `Reset()` operation that unconditionally changes the state of the object to its initial state s_0. An implementation is *sequentially resettable* if any history in which every `Reset()` call is executed in isolation is linearizable. We denote a `Reset()` operation of a sequentially resettable object with `SReset()`. We will refer to an object as writable or resettable if it is a linearizable implementation of a writable or resettable type, respectively.

Our aim is to augment objects with concurrent `Write()` operations, provided that it is possible to write to the objects non-concurrently. More precisely, given a sequentially writable implementation of a writable type $\mathcal{T}$, we show how to implement a linearizable implementation of type $\mathcal{T}$. Throughout this thesis, n denotes the number of processes in the system, indexed from 0 to $n-1$.

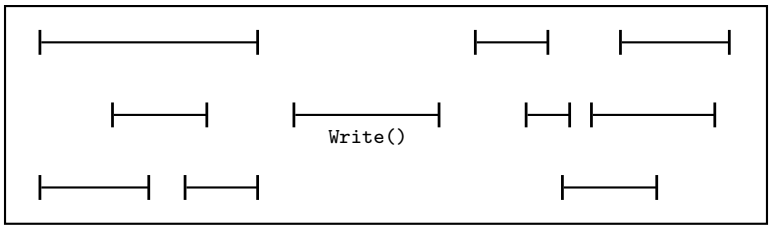

Figure 4.1: An implementation is sequentially writable if every history on that object in which no Write() call overlaps any other operation call is linearizable

4.1 Results and Applications

Our main contribution is the following: Suppose we are provided with a sequentially writable implementation, $\mathcal{SW}$, of a writable type $\mathcal{T}$. We show how to obtain a linearizable implementation $\mathcal{WT}$ of type $\mathcal{T}$ using $O(n^2)$ instances of $\mathcal{SW}$ objects, and $O(n^2)$ additional registers, such that the step complexity of all operations (including `Write()`) changes only by a constant additive term from the corresponding operation on $\mathcal{SW}$.

Our only assumption on $\mathcal{SW}$ is linearizability of executions in which `Write()` operation calls are executed in isolation. The original object $\mathcal{SW}$ can either be provided by the system or be implemented from other, arbitrary, base objects. Our transformation introduces only additional atomic registers. Furthermore, any operation call in the transformed object, including the `Write()` calls even when executed concurrently with any other operation calls, takes the same number of steps (within an additive constant) as the corresponding operation call on $\mathcal{SW}$. The transformation is also deterministic and wait-free, and therefore preserves worst-case step complexity.

Theorem 4.1. *Any sequentially writable object $\mathcal{SW}$ of a writable type can be transformed into a linearizable object $\mathcal{WT}$ of the same type that uses $O(n^2)$ instances of $\mathcal{SW}$ and $O(n^2)$ registers, each of size $\big(2\log n + O(1)\big)$ bits, such that each operation call on $\mathcal{WT}$ has, up to a constant additive term, the same step complexity as the corresponding operation call on $\mathcal{SW}$.*

The transformation is provided in Section 4.3, and the theorem is proved in Section 4.4.2.

To obtain our transformation using a bounded pool of $\mathcal{SW}$ objects, we devise a novel memory management mechanism inspired by Michael's Hazard Pointer technique (Michael, 2004b). This technique can be applied in other domains, such as memory reclamation in concurrent dynamic data structures. In Chapter 7, a more elaborate version of this technique is used to develop a general memory management framework.

Since a `Reset()` operation is a special case of a `Write()` operation, Theorem 4.1 also implies that any sequentially resettable object can be transformed to a resettable one.

In general, for any object that is implemented from writable/resettable base objects (e.g. atomic registers, writable CAS, etc), we can easily add a sequential write/reset (`SWrite()`/`SReset()`) operation by writing to each base object the state that the base object should have after the write/reset operation. The step complexity of such sequential write/reset operation is the sum of step complexities of the write/reset operations on the base objects. Using the result of Theorem 4.1, those sequentially writable/resettable objects can be transformed into the ones that support concurrent write/reset operations.

Let (k,b)-Array denote a resettable type that stores an array S of size k, where each array entry $S[i]$ is of size b bits, for $i \in \{0,\ldots,k-1\}$. This type supports three operations: `Read()`, `Write()`, and `Reset()`. For each $i \in \{0,\ldots,k-1\}$, operation `Read(`i`)` returns the value of $S[i]$, and operation `Write(`i,x`)` writes value x to $S[i]$. Since the type is resettable, a `Reset()` operation resets $S[i]$ to its initial value, for all $i \in \{0,\ldots,k-1\}$. We prove in Section 4.5 that we can implement a sequentially resettable (k,b)-Array, such that reading and writing to each array entry, as well as resetting all array entries can be done in a constant number of steps.

Lemma 4.2. *There is an implementation of a sequentially resettable (k,b)-Array, from $k+1$ registers, each of size $b+O(\log k)$ bits, in which each* `Read()`, `Write()` *and* `SReset()` *operation call has constant step complexity.*

Therefore if an object is implemented only from k read/write registers of size b-bits, then we can replace the registers with the sequentially resettable (k,b)-Array of Lemma 4.2. This way,

the object can be augmented with a sequential reset operation that has optimal step complexity:

Corollary 4.3. *Any object O of some type $\mathcal{T}$ that is implemented from k atomic registers of size b can be transformed into a sequentially resettable object S of the same type that uses $k+1$ registers, each of size $b + O(\log k)$ bits, such that each S.*`SReset()` *requires only a constant number of steps, and any other operation call on S has, up to a constant additive term, the same step complexity as the corresponding operation call on O.*

Our transformations have several interesting applications that we overview in the following.

4.1.1 Multi-Word Registers

We implement a sequentially writable k-word register from k single-word registers $R[0],\ldots,R[k-1]$ as depicted in Figure 4.2. To write the value $(x_0,...,x_{k-1})$, a process writes x_i to $R[i]$ for $i \in \{0,\ldots,k-1\}$ in k consecutive write operation calls, and to read the k-word register, a process reads all registers $R[i]$, $i \in \{0,\ldots,k-1\}$, and returns the k-tuple observed. It is obvious that any history resulting from executing operations on this object is linearizable, provided that each `SWrite()` operation call is not executed concurrently with any other operation call on the k-word register. Applying the transformation of Theorem 4.1, we can obtain a linearizable k-word register from $O(k \cdot n^2)$ linearizable single-word registers, such that each k-word `Read()` and `Write()` operation runs in $O(k)$ steps (which is optimal).

Corollary 4.4. *There is a linearizable implementation of a k-word register from $O(k \cdot n^2)$ single-word registers, such that each* `Read()` *and* `Write()` *operation of the k-word register takes $O(k)$ steps.*

4.1.2 Writable Compare-And-Swap and Fetch-And-Add

Certain primitives, such as atomic compare-and-swap (CAS) or fetch-and-add (FAA), trivially support sequential write operations. For example, to sequentially write the value v to a CAS object, a process can first read its current value x, and then execute `CAS(`x,v`)`. If both steps

shared: `Array<register>` $R[k]$

Operation `SWrite`$(x_0,\ldots,x_{k-1})$

for $i=0$ **to** $k-1$ **do**
  $R[i]$.`Write`(x_i)

Operation `Read()`

for $i=0$ **to** $k-1$ **do**
  $x_i := R[i]$.`Read()`
return $(x_0,\ldots,x_{k-1})$

Figure 4.2: A Sequentially Writable k-Word Register from k Single-Word Registers

are executed in isolation, they must succeed. Similarly, a sequential write of value v to FAA object can be achieved by reading its current value x, and then adding $v-x$ to it. Thus, using our transformation, we can implement writable CAS and writable FAA objects from $O(n^2)$ instances of (non-writable) CAS and FAA objects, respectively, and $O(n^2)$ registers, such that each `Write()` or `CAS()` and `FAA()` operation has constant step complexity, respectively.

4.1.3 Long-Lived Test-And-Set

As another application, randomized one-time test-and-set (TAS) objects implemented from registers can be transformed to resettable (long-lived) ones. We will only give a high level overview of this construction, since we obtain more time and space efficient constructions in Chapter 6.

Given any one-time TAS object implemented from a set $\mathcal{R}$ of registers, we can immediately augment the object with a sequential reset operation to obtain a sequentially resettable TAS object. To sequentially reset the one-time TAS, we simply write the initial value of each register of $\mathcal{R}$ into that register. The step complexity of such a sequential reset operation is proportional to the cardinality of $\mathcal{R}$. Due to the space lower bound by Giakkoupis and Woelfel (2012), $|\mathcal{R}|=\Omega(\log n)$ for any implementation of randomized wait-free one-time TAS from registers. Using the result of Lemma 4.2, we can improve the step complexity of the sequential reset operation to $O(1)$, while using one additional register. Since `Reset()` is a special `Write()` operation, applying the transformation of Theorem 4.1, we can then obtain a resettable (long-lived) TAS object (in which any `Reset()` operation call can overlap other operation calls). This transformation increases the number of registers by a factor of $O(n^2)$, and preserves the (asymptotic) step complexity of `TAS()` and `Reset()` operations. Therefore, a `TAS()` operation on the resulting long-lived TAS

has asymptotically the same step complexity as in the one-time implementation, and a `Reset()` operation requires only a constant number of steps.

Recently, very efficient randomized implementations of one-time TAS objects in the oblivious adversary model were given by Alistarh and Aspnes (2011), Giakkoupis, Helmi, Higham and Woelfel (2013, 2015), and Giakkoupis and Woelfel (2012). For example, Giakkoupis, Helmi, Higham and Woelfel (2015) present a randomized one-time TAS implementation from $O(\log n)$ registers, in which the `TAS()` operation has $O(\log^* n)$ expected step complexity against an oblivious adversary. Applying the transformation described above to this randomized TAS implementations yields a long-lived TAS object that can be reset in constant worst-case steps, and the `TAS()` operation has $O(\log^* n)$ expected step complexity against the oblivious adversary. The resulting implementation uses $O(n^2 \cdot \log n)$ registers.

In Chapter 6, we propose more efficient implementations of long-lived TAS objects.

4.2 Preliminaries

Let $\mathcal{T}$ be a writable type for which we have a sequentially writable implementation. In the following, we refer to instances of this implementation as $\mathcal{SW}$ objects.

4.2.1 Simple Implementation

A straight-forward linearizable implementation $\mathcal{WT}$ of type $\mathcal{T}$, which is wait-free but uses unbounded space follows. The pseudocode is given in Figure 4.3.

A shared register Ptr stores the index i of an entry of an array A of infinitely many $\mathcal{SW}$ objects. The domain of array indices is partitioned into n sets $I_0, \ldots, I_{n-1}$, such that each process p *owns* set I_p of array indices and the corresponding array entries $A[i]$, $i \in I_p$. For example, we can let the domain of indices of A be the set of all integers and $I_p = \{i \geq 0 \,|\, i \bmod n = p\}$, for any $p \in \{0, \ldots, n-1\}$.

In a $\mathcal{WT}$.`Write(`v`)` operation, the calling process chooses an index f that it owns and

shared:
`Array`<$\mathcal{SW}$> $A[0 \ldots \infty]$
`int` $Ptr = 0$

Operation $\texttt{Write}_p(v)$

6 choose a new write-index $f \in I_p$ that p has never chosen before
7 $A[f]$.`SWrite`(v)
8 Ptr.`Write`(f)

Operation $\texttt{Exec}_q$(`Oper`,arg)

9 $i := Ptr$.`Read()`
10 $ret\_val := A[i]$.`Oper`(arg)
11 **return** ret_val

Figure 4.3: Simple Implementation of a Writable Object

that it has never chosen before, and writes v into $A[f]$ by executing $A[f]$.`SWrite`(v). In its last step of its $\mathcal{WT}$.`Write`(v) operation call, the process writes f into register Ptr. During a $\mathcal{WT}$.`Exec`(`Oper`,arg) operation call, where `Oper`(arg) is an operation on $\mathcal{SW}$ other than a `Write()` operation, the calling process p reads the current index i stored in Ptr, and then it executes `Oper`(arg) on $A[i]$. Finally, operation call $\mathcal{WT}$.`Exec`(`Oper`,arg) by process p returns the return value of p's $A[i]$.`Oper`(arg) call.

This way, each `SWrite()` operation call on an object $A[j]$ is executed before j has ever been stored in Ptr, and thus this operation call does not overlap with any other operation call on $A[j]$. Each `Write()` operation call that writes some index i into Ptr linearizes at the point when the write to Ptr happens. Each `Exec()` operation call that reads some index i from Ptr linearizes either when $A[i]$.`Oper`(arg) linearizes, or when Ptr gets written for the first time after p's read of value i from Ptr, whichever comes earlier.

This example is simple, linearizable, and wait-free, however it requires a new $\mathcal{SW}$ object with each `Write()` operation call of object $\mathcal{WT}$: if an unbounded number of `Write()` operation calls get executed, then this implementation requires an unbounded number of $\mathcal{SW}$ objects. Also register Ptr has to store unbounded values.

To fix this problem, we first derive, in the remainder of this section, a template from this

example. We state in Lemma 4.5, a set of sufficient conditions, such that any algorithm following this template and satisfying those conditions is linearizable. Then in Section 4.3, we propose a wait-free linearizable implementation $\mathcal{WT}$ of type $\mathcal{T}$ from a finite number of $\mathcal{SW}$ objects and registers, where each register stores $O(\log n)$ bits. This implementation follows our template. Finally, in Section 4.4, we provide a proof of Lemma 4.5, and then show that $\mathcal{WT}$ satisfies the requirements of that lemma. From that it follows that $\mathcal{WT}$ is linearizable.

4.2.2 A Natural Template

Figure 4.4 provides a template to implement a linearizable object $\mathcal{WT}$ of some writable type $\mathcal{T}$ from sequentially writable objects of the same type. Our goal is then to specify a sufficient condition for linearizability of any implementation following this template.

```
shared:
  Array<SW> A[0...m-1]
  int Ptr = ⊥
/* A and Ptr are only modified where indicated.                    */

Operation Write_p(v)
    ⋮
    choose an appropriate write-index
    f ∈ I_p that is not in use
    ⋮
    A[f].SWrite(v)
    ⋮
    Ptr.Write(f,...)
    ⋮

Operation Exec_q(Op,arg)
    ⋮
    read Ptr and then use the value to help
    determine an appropriate exec-index i
    ⋮
    ret_val := A[i].Oper(arg)
    ⋮
    return ret_val
```

Figure 4.4: Writable Object Template

Similarly to our earlier example, the template uses an array $A[0 \ldots m-1]$, $m \in \mathbb{N} \cup \{\infty\}$, of $\mathcal{SW}$ objects and a shared register Ptr that stores the index i of an array entry of A, and possibly some additional information. We partition set $\{0, \ldots, m-1\}$ into n sets $I_0, \ldots, I_{n-1}$. Each index $i \in I_p$ and its corresponding array entry $A[i]$ is *owned* by process p. An object $A[i]$ is *in use* if and

only if the value of Ptr equals i. Let v_0 be the initial value of type $\mathcal{T}$. We initialize the variables of this template by starting with register Ptr initialized to $\perp$, and each entry of A initialized with v_0, and then having process 0 execute a `Write`(v_0) operation. This initial `Write()` operation call does not overlap any other operation call.

To execute a `Write`(v) operation, the calling process p first chooses an appropriate index $f \in I_p$ such that $A[f]$ is not in use at the point of invocation of the `Write()` operation call (what "appropriate" means, depends on the actual implementation). We call index f a *write-index*. Then process p executes $A[f]$.`SWrite`(v), and later it *puts* $A[f]$ *into use* by writing f (and possibly some augmenting information) to Ptr. If $A[f']$ was in use immediately before $A[f]$ is put into use, for some index $f' \neq f$, then we say object $A[f']$ gets *retired* at that point. These are the only ways array A and Ptr can be changed by process p.

In an `Exec`(`Oper`,*arg*) operation E, the calling process q first determines an index i, which must satisfy certain requirements (see Lemma 4.5 below). We call index i an *exec-index*. Then process q calls $A[i]$.`Oper`(*arg*), and returns the return value of that operation call. Process q can only access A through this one operation, and it is not allowed to change Ptr. Implementations following this template differ only in how the write- and exec-indices are chosen, and the additional mechanisms needed to support these choices.

Lemma 4.5. *An implementation of* `Write()` *and* `Exec()` *based on the template in Figure 4.4 is linearizable if it guarantees the following:*

(C) For each `Exec()` *operation call E that invokes an* `Oper()` *operation Op on $A[i]$, for any i, there is a point t^* between the invocation of E and the invocation of Op, such that*

(C1) $A[i]$ is in use at point t^*, and

(C2) no `SWrite()` *operation call on $A[i]$ overlaps with interval $[t^*, t_{Op@rsp}]$.*

A complete proof of this Lemma is given in Section 4.4.1. In Section 4.3, we propose a wait-free implementation of a writable object from a bounded number of sequentially writable

objects of the same type, which follows this template, and satisfies the properties of this lemma (as we prove in Section 4.4.2), and so is linearizable.

4.2.3 Using Hazard Pointers to Bound the Number of Memory Locations

In this section, we show what we can achieve if we use standard memory reclamation techniques like Hazard Pointers by Michael (2004b), or Pass-the-Buck technique by Herlihy, Luchangco and Moir (2002) to bound the number of sequentially writable objects of our simple implementation described in Section 4.2.1. Even though the resulting implementation uses a bounded number of sequentially writable objects, it guarantees only lock-freedom, and requires unbounded sequence numbers which implies registers of unbounded size.

A common technique in shared memory algorithms is using an *announce array* . Each process p informs other processes about the task it intends to perform by writing to a single-writer multi-reader register $Annc[p]$. This technique was originally used by Herlihy (1988) in his wait-free universal construction, where each process p announces the operation it intends to execute, which allows other processes to "help" p finish its operation call in a wait-free manner. Hazard Pointers, introduced by Michael (2004b), form an announce array in which processes announce their possible accesses to shared memory locations.

The implementation described below is a straight-forward application of the Hazard Pointer technique (Michael, 2004b) to our unbounded space implementation from Section 4.2.1. In addition to showing the limitations of the Hazard Pointer technique (no wait-freedom and no bound on the size of registers), this description also serves as an introduction to our wait-free implementation in Section 4.3.

To bound the size of array A in our example from Section 4.2.1, we need to reuse, during `Write()` operation calls, some of the previously used and then retired objects of A. The main challenge is that during a `Write()` operation, the calling process p has to identify a write-index f of an object $A[f]$ (that was possibly in use previously but is now retired) such that no process will access $A[f]$ before object $A[f]$ has been put back into use.

We use an array A of size $n(2n+1)$ of $\mathcal{SW}$ objects. Each process p owns the set $I_p = \{i \mid 0 \leq i < n(2n+1)\ ,\ i \bmod n = p\}$ of $2n+1$ indices. Moreover, each process p keeps track of a local sequence number ctr_p, which increments modulo n with every `Write()` operation call. Process p also maintains a local list $usedlist_p$ of indices that it cannot reuse yet. Register Ptr now stores a pair (i,c), where i is an index of array A, and c is a sequence number used by the algorithm to ensure that throughout any execution, the same value (i,c) does not get written twice to Ptr during n consecutive `Write()` operations.

In an `Exec(Oper,`arg`)` operation call, a process q first reads a pair (i,c) from Ptr. Then it *announces* index i by writing it to a single-writer multi-reader register $Annc[q]$. Array element $Annc[q]$ effectively acts as the hazard pointer process q holds, suggested by Michael (2004b). After announcing its possible access of $A[i]$, process q reads Ptr again. If Ptr has changed since q read it for the first time during its `Exec()` call, q starts its `Exec()` operation call over. Otherwise, it uses i as its exec-index, by calling $A[i]$.`Oper(`arg`)` and returning the result.

To execute a `Write(`v`)` operation call, a process p first chooses a write-index $f \in I_p$ that is not in $usedlist_p$. After that, p adds f to its $usedlist_p$, executes $A[f]$.`SWrite(`v`)`, and then it writes the pair (f,ctr_p) to Ptr. Next p increments its local sequence number ctr_p modulo n. If after this increment $ctr_p = 0$, then p reads the entire announce array, and removes any index from $usedlist_p$ that p did not read in this last scan of the announce array. Since this is done once every n `Write()` operation calls by p, $usedlist_p$ can store at most $2n$ indices. Thus, $|I_p| = 2n+1$ guarantees that p can always find a write-index that it owns but is not in $usedlist_p$ during a `Write()` operation call.

This implementation follows the template of Section 4.2.2. Now we argue that it also satisfies the properties of Lemma 4.5, and so is linearizable. First, the announce array guarantees the following: If the object corresponding to an index $i \in I_p$ is in use at some point t, and $Annc[q] = i$ throughout the interval $[t,t']$ for some $t' > t$, then process p will not choose i as a write-index during interval $[t,t']$, and, in particular, no $A[i]$.`SWrite()` operation call will overlap with the

interval $[t,t']$.

Second, in an `Exec(Oper,`arg`)` operation call E, a process q only executes $A[i]$.`Oper(`arg`)` if it read the same value (i,c) from Ptr twice, and announced index i in-between those two reads. This ensures that $A[i]$ is in use at the point of the announcement. Choose t^* to be this point, thus satisfying condition (C1) of Lemma 4.5. Hence, process p, the owner of index i, will not choose i as a write-index again until q has announced a different index (overwritten $Annc[q]$ with a value different from i), and this does not happen during E. Thus, it follows that the interval starting with t^* and ending with the response of q's $A[i]$.`Oper(`arg`)` operation call does not overlap with any `SWrite()` operation call on $A[i]$. This satisfies condition (C2) of Lemma 4.5.

4.3 Wait-Free Implementation

Let $\mathcal{T}$ be a writable type, and assume we are equipped with sequentially writable objects of type $\mathcal{T}$. In the following, we refer to instances of this object as $\mathcal{SW}$ objects. We described a simple wait-free implementation of type $\mathcal{T}$ in Section 4.2.1. It uses an unbounded number of $\mathcal{SW}$ objects, and one register of unbounded size. Then we outlined how we can apply the Hazard Pointer technique to bound the number of base objects, sacrificing wait-freedom, and still requiring registers of unbounded size. It is challenging to bound the number of required base objects, and the size of each register, while maintaining wait-freedom, even if we are not concerned about the step complexity. We now propose a wait-free and linearizable implementation $\mathcal{WT}$ of type $\mathcal{T}$ that uses $O(n^2)$ instances of $\mathcal{SW}$ and $O(n^2)$ registers of size $O(\log n)$. The step complexities of the operations on the resulting object are at most a constant additive term more than the step complexities of the corresponding operations on the sequentially writable object. The proposed algorithm follows.

4.3.1 High Level Idea

Our implementation follows the template in Figure 4.4. The pseudo-code of the algorithm can be found in Figure 4.5 on page 49. All claims referred to in this section appear in Section 4.4.

Process p's main goal during its `Write()` operation call is to find a write-index $f \in I_p$ of an object $A[f]$ that is currently not in use, and that no process will access before p puts it into use. In an `Exec(Oper,`arg`)` operation, the calling process q has to identify an exec-index i and calls $A[i]$.`Oper(`arg`)` such that condition (C) of Lemma 4.5 is satisfied.

In this implementation, similar to the one in Section 4.2.3, we use an announce array A of size n. During an `Exec()` operation call, process q reads a potential exec-index i from Ptr, and then announces its intention to access the object $A[i]$ by writing i to its designated array element $Annc[q]$. Then process q has to ensure that its safe to access this object. In the implementation of Section 4.2.3, process q reads Ptr again, and unless it reads the same index i from Ptr, it does not access object $A[i]$. This is because reading the index, announcing it, and accessing the corresponding object are separate shared memory steps, and so it is possible that it takes process q a "long" time to execute all these steps, and therefore q's announcement might not get seen in time.

Achieving Wait-Freedom. A helping technique is typically used to assist in achieving wait-freedom (Fatourou and Kallimanis, 2011; Herlihy, 1988, 1990, 1991; Kogan and Petrank, 2011, 2012; Plotkin, 1989; Timnat, Braginsky, Kogan and Petrank, 2012; Timnat and Petrank, 2014). In this mechanism, faster processes help slower ones, by performing some work for them, in addition to their own. Recently Censor-Hillel, Petrank and Timnat (2015) presented one possible formal definition of a helping mechanism. In our implementation, we devise a new helping mechanism, which allows each writer to provide an alternative exec-index called a "hint" to a process q, which q uses instead of the one it first chose, in the event that q took too long to make its announcement during an `Exec()` operation call.

More specifically, suppose p writes an index $i \in I_p$ to Ptr at some point t. Our main idea,

which allows a wait-free implementation is that the owner, p, of index i, gives a deadline to any process q that reads i from Ptr after t, to announce index i. If q announces i before this deadline, then p guarantees that it sees q's announcement of i, and it does not use i as a write-index again, until q announces another object. However, if q does not announce i before that deadline, then q will use an alternative exec-index j that p provides to q. In this case, process p guarantees that $A[j]$ was in use at some point during q's `Exec()` call, and also p does not use j as a write-index again until q's `Exec()` call responds, so that conditions of Lemma 4.5 are satisfied.

To achieve those two guarantees, process p has to

(I) read the entire announce array after the deadline, and before it uses i as a write-index again after t (that is the point when p writes i to Ptr), and

(II) provide an alternative exec-index, called a *hint*, to all other processes before the deadline.

Let q be a process that reads i from Ptr after p writes i to this register at t. We choose the deadline for process q to announce index i, to be the first point after t, where p provides a hint to q. Therefore, whenever process p writes an index i to Ptr at t, then p does not use i again as a write-index until p reads the entire announce array after it provides a hint to all processes individually.

Hint Mechanism. By (II), once p writes an index i into Ptr, it has to provide an alternative exec-index, which we call a hint, to each process before the deadline for announcing i. This way if a process q does not announce i before the deadline, then there is a hint available for q. A hint j that is provided by process p to q has to be a valid exec-index, i.e. $A[j]$ must have been in use at some point during q's current `Exec()` operation call, and also index j must not get reused before q's `Exec()` completes (so that the conditions of Lemma 4.5 are satisfied).

To achieve this, each process p employs a *hint array* $H'_p[0 \ldots n-1]$, and a single-writer multi-reader register S_p. Register S_p stores a sequence number that p increments at the beginning of every `Exec()` operation call. For now, assume the sequence number S_p is unbounded; We

will discuss later how to bound it. Array entry $H'_p[q]$ is used to store a *hint* that process p can provide to process q during a `Write()` operation call.

Consider a `Write()` operation call by p with the write-index f, and suppose p wants to provide a hint to process q during this operation call. To that end, p first reads q's sequence number S_q into a local variable s, then it checks whether $H'_p[q]$ already contains a pair $(\cdot,s)$. If yes, then there is already a valid hint stored in $H'_p[q]$ that q might be using as its exec-index. So p does not provide a new hint, and it only writes f into Ptr. Otherwise, p provides hint f with sequence number s to process q by first putting object $A[f]$ into use, and then writing the pair (f,s) to $H'_p[q]$. Process p must not use index f as a write-index again as long as a pair $(f,\cdot)$ is stored in H'_p. Thus p scans hint array H'_p with each scan of the announce array, and it avoids choosing a write-index from indices that it finds in either of those two arrays.

During an `Exec()` operation call E, the calling process, q, first increments S_q. Let s^* be the value q writes to S_q during E. Then, q reads an index $i \in I_p$ from Ptr, and announces this index. In order to check if q made the announcement before the deadline for index i, it checks if there is a hint from p, the owner of index i, with sequence number s^* in $H'_p[q]$. If yes, then q uses the index in $H'_p[q]$ as its exec-index, and otherwise, process q knows that its announcement was made before the deadline for i, and so it uses i as its exec-index.

Deamortization. By (I) and (II), after putting an object $A[i]$ in use, p has to first provide a hint to all other processes and then read the entire announce arrays before it can reuse index i. This will result in executing $\Omega(n)$ steps per each `Write()` operation call. By allowing a bigger set of indices, I_p, for each process p, we can have one deadline for every n objects that p puts into use. This way it is enough to provide hints and then read the announce array every n `Write()` operation calls, and so we can achieve a constant amortized step complexity. To achieve even more efficient `Write()` operations, we deamortize this work over a sequence of $2n$ `Write()` operation calls. Therefore, in each of its `Write()` operation calls, p provides a hint to one of the processes in the system, and reads one element from the announce array $Annc$ and the hint array

H'_p in a round-robin fashion. Hence, once p puts an object $A[i]$ into use, it takes this process n `Write()` calls until it provides a hint to each process, and then another n `Write()` calls to read the entire announce and hint arrays. Therefore, p does not use index i again as its write-index until it executes at least $2n$ additional `Write()` operation calls since it last set $Ptr = i$.

Since it takes process p some time to read the entire array $Annc$, the information it reads might be outdated. To deal with this, we benefit from the following observation: Even though process p might read an outdated announcement from $Annc[q]$, process q can only have updated its announcement to indices of objects that were in use since p read $Annc[q]$ last. Thus, any index $i \in I_p$ that p did not find in its last read of announce array entries $Annc[0],\ldots,Annc[n-1]$ and hint array entries $H'_p[0],\ldots,H'_p[n-1]$, and it did not write into Ptr during its last $2n$ `Write()` calls is an appropriate write-index.

Why it Works. Consider an `Exec()` operation call E by some process q, and let $S_q = s$ right after q increments this variable at the beginning of E. We argue that if q invokes an $A[i]$.`Oper()` call during E, for some $i \in I_p$, the conditions of Lemma 4.5 are satisfied.

First Suppose q uses i as its exec-index, because it reads (i,s) from its hint array entry $H'_p[q]$, for some p. Since q increments its sequence number at the beginning of E, process p must have read s from S_q and written (i,s) to $H'_p[q]$ after q invokes E and before q reads $H'_p[q]$. Therefore, p must have also put $A[i]$ into use at some point t^* after the invocation of E and before q reads $H'_p[q]$, hence condition (C1) is satisfied. Moreover, process p does not choose its write-index from indices that are stored in entries of H'_p and $Annc$. Thus, this process does not use i again as a write-index in any subsequent `Write()` operation call, as long as the pair (i,s) remains in $H'_p[q]$. Process p does not change the value stored in $H'_p[q]$ until p reads a sequence number other than s from S_q. Thus, $H'_p[q] = (i,s)$ at least until q's operation call $A[i]$.`Oper()` during E responds. This implies that between t^* and the response of q's $A[i]$.`Oper()` call, process p does not use i as its write-index again, and therefore it does not execute any $A[i]$.`SWrite()` operation call in this interval. Hence, condition (C2) of Lemma 4.5 is also satisfied.

Next suppose q uses i as its exec-index because it reads i from Ptr, where $i \in I_p$, and some pair (j,s') from $H'_p[q]$ during E, where $s' \neq s$. Condition (C1) is clearly satisfied, as q reads $Ptr = i$ between the invocation of E and the invocation of its $A[i]$.`Oper()` operation call. Let t be the point at which q reads index i from Ptr, and let t' be the point when q announces that index during E. Process p completes at most $n-1$ `Write()` operation calls during $[t,t']$. Because otherwise, p would have provided a hint with sequence number s to q in this interval. This implies that q announces i before its deadline. Since p has to execute at least $2n$ `Write()` operation calls after t and before it can use i as a write-index again, and it has completed fewer than n of them during $[t,t']$, in its at least next n `Write()` operation calls after t', process p reads q's announcement. Therefore, p cannot use i as its write-index, before q announces another index, which can only happen after q's current `Exec()` is completed. Thus, condition (C2) is also satisfied in this case.

Bounding the Sequence Number. In order to bound the sequence number stored in S_q, we increment S_q modulo $2n$. This creates the following problem: Suppose q writes some value s to S_q at the beginning of some `Exec()` call E, and later during E process q finds a pair (f,s) as a hint in $H'_p[q]$. If S_q is unbounded (and strictly increases with each `Exec()` call), then q can ensure that process p, which wrote this pair to $H'_p[q]$, read value s from S_q and wrote f to Ptr both during E, which guarantees condition (C1) of Lemma 4.5. But by incrementing S_q only modulo $2n$, this guarantee is lost.

To provide the same guarantee with bounded sequence numbers, we apply two tricks. First, we maintain an additional hint array $H_p[0 \ldots n-1]$ for every process p. Each time p wants to provide its write-index f as a hint to q, this process writes to $H_p[q]$ before writing f to Ptr, and then it writes to $H'_p[q]$ after updating Ptr. Second, we ensure that during any sequence of $2n$ `Exec()` calls, process q resets each hint array entry $H_r[q]$ and $H'_r[q]$, $r \in \{0,\ldots,n-1\}$, by writing $(\perp,\perp)$ to it, at least once.

These two together ensure that if S_q has wrapped around since p read some value s from

this register, then at least one of $H_p[q]$ and $H'_p[q]$ is reset to $(\perp,\perp)$. Thus to prevent q from using an outdated hint during an `Exec()` operation call E, this process only uses a hint f as its exec-index, if it reads the same pair (f,s) from both $H_p[q]$ and $H'_p[q]$, where s is the current value of S_q. This way, our algorithm guarantees that (1) p wrote this pair to $H_p[q]$ and $H'_p[q]$ both during one `Write()` call, and (2) both writes to hint entries happen after q updates its S_q at the beginning of E and before q reads them. Thus, the step in which p writes f to Ptr also happens during the same interval, which guarantees condition (C1) of Lemma 4.5. See Claim 4.25 for the full proof.

Memory Management. We equip our algorithm with a memory management technique that allows a process p in each `Write()` operation call to *recycle* objects, i.e., to identify a write-index $f \in I_p$ such that no other process is concurrently using or is poised to use f as an exec-index. Our technique is similar to that of Hazard Pointers (Michael, 2004b), but more efficient with respect to step complexity: The deterministic version of the Hazard Pointers technique has $O(\log n)$ amortized step complexity per recycled object, and the randomized version has expected $O(1)$ amortized step complexity. Our technique allows us to deterministically recycle one object in constant worst-case time. We believe that the technique is interesting by itself. It has already found other applications (Brown, Ellen and Ruppert, 2013, 2014; Ellen and Woelfel, 2013; Giakkoupis and Woelfel, 2014). Later in Chapter 7, we extend the ideas used here to a stand-alone memory reclamation technique.

When a process p decides on a write-index f in a `Write()` operation call, it chooses one in I_p that (1) is currently not announced, (2) is not stored in the hint arrays H_p and H'_p, and (3) has not been used as a write-index in any of p's $2n$ preceding `Write()` operation calls. Process p stores indices satisfying these properties in a local set $free_p$.

To maintain $free_p$, process p uses a helper operation `recycle()`. This operation is called exactly once during every `Write()` operation call, and returns an index f that p owns and can be used as a write-index for the calling `Write()` operation. We describe the `recycle()` operation

in more detail in Section 4.3.2.

4.3.2 Detailed Description

We now describe the `Write()` and `Exec()` operations of our writable object $\mathcal{WT}$ of type $\mathcal{T}$ in detail. Pseudocode is given in Figure 4.5. Shared variables and public operations of the writable and sequentially writable objects and registers start with an upper-case character, and local variables and internal operations start with a lower-case character.

Figure 4.5 does not represent the initial state of the object, but rather the initial state is obtained by starting with the initial state depicted in Figure 4.5, and then having process 0 execute a `Write(`v_0`)` operation call, where v_0 is the initial state of the type. Clearly, one may easily obtain an implementation that does not require this assumption, by initializing all objects and variables to the state they are in immediately after such an initial `Write(`v_0`)` operation call. We make this assumption to be compatible with the template of Section 4.2.2, and also to avoid having different initial values for local variables of different processes in the pseudocode of Figure 4.5.

We use a sufficiently large array A of instances of the sequentially writable object $\mathcal{SW}$. Let $\mathcal{I}$ denote the set of indices of A. Each process p owns a set $I_p \subseteq \mathcal{I}$, of indices, where $I_p = \{i \in \mathcal{I} \,|\, i \bmod n = p\}$. We chose $\mathcal{I} = \{0,\ldots,n(8n+9)-1\}$. With some additional care, it may be possible to reduce the size of this set by a small constant factor. We did not attempt to do so, in order to not distract from the core ideas of this algorithm. We assume a helper operation `owner(`i`)`, which returns for $i \in \mathcal{I}$ the process p, such that $i \in I_p$.

We use an announce array $Annc[0\ldots n-1]$, and for each process p there are two hint arrays $H_p[0\ldots n-1]$ and $H'_p[0\ldots n-1]$. Only process p writes to register $Annc[p]$, and only processes p and q read from and write to $H_p[q]$ and $H'_p[q]$. Each element of H_p and H'_p is initialized to $(\perp,\perp)$, and each element of $Annc$ to $\perp$. A shared register Ptr, which is initially $\perp$, stores the index of the object in use. For each process p, the shared register S_p stores a sequence number. Each register needs to store at most $O(n^2)$ different values, and therefore registers of

```
shared:
  Array<SW> A[0...n(8n+9)-1] = (v_0,...,v_0)
  Array<int> Annc[0...n-1] = (⊥,...,⊥)
  int Ptr = ⊥
  for i ∈ {0,...,n-1}:
    Array<int,int> H_i[0...n-1] = ((⊥,⊥),...,(⊥,⊥))
    Array<int,int> H'_i[0...n-1] = ((⊥,⊥),...,(⊥,⊥))
  int S_0,...,S_{n-1} = -1
local:
  Queue usedQ_p, anncQ_p, hintQ_p, hint'Q_p
  // 2n+2 ⊥ values are initially in each queue
  Set<int> free_p = I_p
  Array<int> c_p[0...8n+8] = (0,...,0)
  int lptr_p = ⊥
  int ctr_p = 0
```

```
/* v_0 is the initial value
of type T.  The initial
state of the object is
obtained by starting with
all variables initialized
as depicted, and then
having process 0 execute
a Write(v_0) operation.

I = {0,...,n(8n+9)-1}
I_p = {i ∈ I | i mod n = p}
|I_p| = 8n+9                    */
```

Operation $\texttt{Write}_p(v)$

```
12 f = recycle_p()
13 A[f].SWrite(v)
14 s := S_{ctr_p}.Read()
15 (·,olds) := H_p[ctr_p].Read()
16 (·,olds') := H'_p[ctr_p].Read()
17 if s ≠ olds ∨ s ≠ olds' then
18     H_p[ctr_p].Write(f,s)
19     s' := S_{ctr_p}.Read()
20     Ptr.Write(f)
21     H'_p[ctr_p].Write(f,s')
22 else
23     Ptr.Write(f)
24 ctr_p := (ctr_p + 1) mod n
```

Operation $\texttt{recycle}_p()$

```
25 f := free_p.remove()
26 updateQ_p(usedQ_p,f)
27 a := Annc[ctr_p].Read()
28 updateQ_p(anncQ_p,a)
29 (h,.) := H_p[ctr_p].Read()
30 updateQ_p(hintQ_p,h)
31 (h',.) := H'_p[ctr_p].Read()
32 updateQ_p(hint'Q_p,h')
33 return f
```

Operation $\texttt{Exec}_q(\texttt{Oper},arg)$

```
34 x := (S_q.Read() + 1) mod 2n
35 S_q.Write(x)
36 i := Ptr.Read()
37 Annc[q].Write(i)
38 p := owner(i)
39 (j,s) := H_p[q].Read()
40 (j',s') := H'_p[q].Read()
41 if s = s' = x ∧ j = j' then
42     i := j
43 ret_val := A[i].Oper(arg)
44 H_{x mod n}[q].Write(⊥,⊥)
45 H'_{x mod n}[q].Write(⊥,⊥)
46 return ret_val
```

Operation $\texttt{updateQ}_p(\texttt{Queue}\ qu,\ \texttt{int}\ x)$

```
47 if p = owner(x) then
48     c_p[x] := c_p[x] + 1
49     qu.enq(x)
50 else
51     qu.enq(⊥)
52 y := qu.deq()
53 if y ≠ ⊥ then
54     c_p[y] := c_p[y] - 1
55     if c_p[y] = 0 then free_p.add(y)
```

Figure 4.5: Implementation of the Writable Object

size $2\log n + O(1)$ bits suffice.

Our implementation provides, in addition to the operations `Write()` and `Exec()`, helper operations `recycle()` and `updateQ()` that are used only internally and take care of memory reclamation.

The Write() Operation. To execute `Write(v)`, process p has to find a write-index f of a free $\mathcal{SW}$ object in A, to which it can write using an `SWrite()` operation call, and that it can then put into use. To do so, process p calls `recycle()` (line 12), which returns the index f that p will subsequently use as a write-index. In line 13, p writes the value v to $A[f]$, by executing `SWrite(v)`. Then in line 14, p reads the sequence number stored in S_{ctr_p} into a local variable s. (Recall that ctr_p is a local variable that gets incremented modulo n at the end of the `Write()` operation call, and serves as a pointer to process q.) In lines 15–16, p reads the second component of $H_p[ctr_p]$ and $H'_p[ctr_p]$ into $olds$ and $olds'$, respectively, to determine the sequence numbers associated with the hints currently stored in those registers. If $s = olds = olds'$, then that hint is valid for the process with identifier ctr_p, and p does not need to provide a new hint. Instead, it can simply put $A[f]$ into use by writing f to Ptr (in line 23).

Otherwise, the current hint is not valid, so in lines 18–21 process p attempts to provide a new hint to the process with identifier ctr_p. First, p writes the value (f,s) into $H_p[ctr_p]$. Then p reads again the current sequence number of the process with identifier ctr_p into s', because that value may have changed since p read it last. Next, p updates Ptr by writing value f into it, and finally p writes (f,s') into $H'_p[ctr_p]$. (For the sake of simplicity, we allow process p to provide a hint to itself, when $ctr_p = p$.)

At the end of the `Write()` operation call, in line 24, p increments the value of ctr_p (modulo n). This ensures that in n consecutive `Write()` operation calls, p attempts to provide one hint to each process.

The Exec() Operation. Consider a call of `Exec(Oper,`arg`)` by process q. In lines 34–35, process q first increments modulo $2n$ the value of its sequence number S_q. Then in lines 36–38,

q reads the index of the object that is in use from Ptr into a local variable i, announces i by writing it into $Annc[q]$, and determines the owner p of object $A[i]$. In lines 39–40, process q reads the hint array entries $H_p[q]$ and $H'_p[q]$ into (j,s) and (j',s'), respectively. If it determines that the hint is valid, i.e., $(j,s) = (j,s')$ where s is the last value q wrote into S_q, then q replaces index i with j (lines 41–42). Now, i is the exec-index that q will use, and so in line 43, q executes operation `Oper(`arg`)` on object $A[i]$. Finally, in lines 44–45 it writes $(\perp,\perp)$ to registers $H_{x \bmod n}[q]$ and $H'_{x \bmod n}[q]$, where x is the current value of S_q, to prevent itself from using a very old hint in future `Exec()` operation calls. The return value of this operation call is the value returned from the execution of $A[i]$.`Oper(`arg`)`.

The recycle() and updateQ() Operations. Operation `recycle()` takes no input, and it returns the index of an object that p owns and which it can use as a write-index. The `recycle()` operation does not write to any shared objects. Most of the objects it uses are local to the calling process, p. In particular, it uses four local queues $usedQ_p$, $anncQ_p$, $hintQ_p$, and $hint'Q_p$, and a local set data structure $free_p$. We assume that the data structure $free_p$ provides the operations `add()` and `remove()`, where $free_p$.`add(`i`)` adds value i to $free_p$, if i is not already in the set, and $free_p$.`remove()` removes one (arbitrary) element from $free_p$ and returns it. Since the domain of values stored in $free_p$ is I_p, which is of size $O(n)$, such a set data structure can be easily implemented from an array and a linked list in $O(n)$ space and with constant worst-case running time per operation.

The queues keep track of indices that p cannot use as write-indices. Initially, each queue contains $2n+2$ $\perp$-values, and $free_p$ contains all indices in I_p. The algorithm maintains the invariant that at each point, each index $i \in I_p$ is either in one of p's queues or it is in $free_p$ (see Claim 4.13). A local counter array c_p keeps track for each index $i \in I_p$, how many copies of i appear in p's queues (see Claim 4.12). Operation `updateQ`$_p()$ maintains those queues and counter array c_p. First we explain the implementation of `updateQ()` and then we discuss how `recycle()` works.

Operation $\texttt{updateQ}_p()$ takes two inputs, one of p's local queues qu and an index x, and executes no operation on any shared object. Process p enqueues exactly one, and dequeues exactly one element from qu in this operation call, hence the size of qu remains unchanged at the end of this operation call. Process p modifies its queues only in this operation, and since each of these queues initially contains $2n+2$ elements, the size of each remains $2n+2$ (see Claim 4.11).

During this operation, process p checks whether $x \in I_p$ (i.e. p owns x). If so, then p increments $c_p[x]$ by one, and enqueues x into qu. Otherwise, it only enqueues $\perp$ into qu (in order to maintain the size of the queue unchanged at the end of each call of this operation). Then p dequeues one element y from qu. If y is an index (i.e. it is not equal to $\perp$), then it is an index that p owns (see Claim 4.9), so p decrements $c_p[y]$ and checks if $c_p[y] = 0$. If so, then index y is not in any of p's queues (see Claim 4.12), and p adds y to free_p.

To execute `recycle()`, a process p first removes an arbitrary index f from free_p in line 25. (Claim 4.14 guarantees that there is at least one element in free_p before p executes this line.) Then it updates usedQ_p by executing $\texttt{updateQ}(\mathit{usedQ}_p, f)$. Note that f must be in I_p (see Claim 4.9), thus, p enqueues f into usedQ_p during this operation call. This implies that before p puts object $A[f]$ into use, it enqueues f into usedQ_p. Next process p reads $\mathit{Annc}[\mathit{ctr}_p]$, and the first elements of $H_p[\mathit{ctr}_p]$ and $H'_p[\mathit{ctr}_p]$ into a, h, and h', and executes $\texttt{updateQ}(\mathit{anncQ}_p, a)$, $\texttt{updateQ}(\mathit{hintQ}_p, h)$, and $\texttt{updateQ}(\mathit{hint'Q}_p, h')$, respectively. (Again, for the sake of simplicity, we allow p to read its own announcement and the hints p provided to itself. Even though this is not necessary.) Hence, p enqueues indices from I_p that it reads during its lazy scan of Annc, H_p, and H'_p into its corresponding queues. With each execution of `updateQ()`, p also dequeues one element from the corresponding queue.

Now we explain why any index in free_p can be returned from `recycle()` to be used as a write-index by p. (For a full proof see Section 4.4.) Suppose p dequeues y from a queue qu during a `recycle()` operation call R, and let $R_0, \ldots, R_{2n+1}$ be the last $2n+2$ `recycle()` calls

p executes before R. Then the implementation of `updateQ()` guarantees that this copy of y was enqueued into qu during R_0 (if $qu = usedQ$, this implies that y was the write-index of the corresponding `Write()` operation call) and remained in qu throughout $R_1, \ldots, R_{2n+1}$ (see Claim 4.15). Process p then adds y to $free_p$ if no instance of y is in any of its queues. This implies that y is not added to any of p's queues during the current and the last $2n+1$ `recycle()` calls. Therefore, y is not used as a write-index during the current and the last $2n+1$ `Write()` operation calls (i.e. $2n+2$ `Write()` operation calls), and was not read from $Annc$, H_p and H'_p during the last $2n+2$ `recycle()` operation calls. Therefore, any index in $free_p$ can be used as a write-index for p.

4.4 Analysis and Correctness

In this section, we first prove the correctness of Lemma 4.5 from Section 4.2.2. This lemma states that if an implementation based on the template in Figure 4.4 has some specific properties, then it is linearizable. Then in Lemma 4.8, we show that the implementation of Figure 4.5 satisfies those properties, and hence, is linearizable.

4.4.1 Proof of Lemma 4.5

Lemma 4.5 states that an implementation of an object $\mathcal{WT}$ based on the template given in Figure 4.4 is linearizable if the implementation satisfies the following condition:

(C) For each `Exec()` operation call E that invokes an `Oper()` operation Op on $A[i]$, for any i, there is a point t^* between the invocation of E and the invocation of Op, such that

(C1) $A[i]$ is in use at point t^*, and

(C2) no `SWrite()` operation call on $A[i]$ overlaps with interval $[t^*, t_{Op@rsp}]$.

The remainder of this section provides the complete proof of this lemma.

Consider any transcript Λ obtained from an implementation based on the template given in Figure 4.4, such that it satisfies condition (C) of Lemma 4.5. To simplify the linearizability proof of any transcript that starts from the initial state, we assume that the initial state is achieved by initializing the shared variables $Ptr = \perp$ and $A[i] = v_0$, for all i, and executing a `Write`(v_0) by process 0 in isolation.

By (C2), no `Oper()` operation call on an object $A[i]$ overlaps with any `SWrite()` operation call on the same object. Moreover, only the process that owns i executes $A[i]$.`SWrite()`. Hence, no two `SWrite()` operation calls on $A[i]$ overlap. Therefore, for any i

$$\text{no } A[i].\texttt{SWrite()} \text{ operation call overlaps any other operation call on } A[i]. \tag{4.1}$$

Consider an operation call M on object $\mathcal{WT}$ in Λ. Operation call M is *matured* if M is a `Write()` call during which the calling process writes to Ptr, or if M is an `Exec()` call in which an $A[i]$.`Oper()` call, for some exec-index i, is invoked.

Let t_0 denote the point when the execution of Λ starts. We associate a timestamp $\texttt{TS}_{Ptr}(t)$ with each point t, where $\texttt{TS}_{Ptr}(t) = s$, if in $[t_0, t]$ Ptr gets written s times. Thus,

$$\texttt{TS}_{Ptr} \text{ is a non-decreasing function.} \tag{4.2}$$

For a matured $\mathcal{WT}$.`Write()` operation call M, executed by some process p, during which p writes index i into Ptr, we let $t(M)$ be the point immediately after p writes i to Ptr. For a matured $\mathcal{WT}$.`Exec()` operation call M, in which some $A[i]$.`Oper()` operation call is invoked, for some exec-index i, we let $t(M)$ be the latest point between the invocation of M and the invocation of $A[i]$.`Oper()`, such that (C1) and (C2) are satisfied for $t^* = t(M)$. Hence by (C1), $Ptr = i$ at point $t(M)$. For any matured operation call M, we assign a timestamp $\texttt{TS}(M) = \texttt{TS}_{Ptr}(t(M))$.

Therefore, for any matured `Write()` or `Exec()` operation call M by a process p, which either

writes i to Ptr or invokes an $A[i]$.`Oper()` operation call respectively, we have

$$\begin{aligned} &t(M) \in [t_{M@inv}, t_{M@rsp}], \\ &Ptr = i \text{ at } t(M), \text{ and} \\ &\texttt{TS}(M) = \texttt{TS}_{Ptr}(t(M)). \end{aligned} \tag{4.3}$$

Claim 4.6. *Let M_1 and M_2 be two distinct matured operation calls on $\mathcal{WT}$ in Λ, and suppose some operation call Op_1 on a sequentially writable object $A[i_1]$ is invoked during M_1, and some operation call Op_2 on $A[i_2]$ is invoked during M_2, for some i_1 and i_2. If $\texttt{TS}(M_1) = \texttt{TS}(M_2)$, then $i_1 = i_2$.*

Proof. Assume w.l.o.g. that $t(M_1)$ precedes $t(M_2)$. According to (4.3), $\texttt{TS}(M_1) = \texttt{TS}(M_2)$ implies $\texttt{TS}_{Ptr}(t(M_1)) = \texttt{TS}_{Ptr}(t(M_2))$, so no process writes to Ptr between $t(M_1)$ and $t(M_2)$. By (4.3), $Ptr = i_1$ at $t(M_1)$ and $Ptr = i_2$ at $t(M_2)$, hence, $i_1 = i_2$. □

Let $\mathcal{O}_\ell$ denote the set of all matured operation calls M on $\mathcal{WT}$ in Λ, where $\texttt{TS}(M) = \ell$, for $\ell \in \mathbb{N}$. By Claim 4.6, there exists an index i_ℓ, such that all operation calls in $\mathcal{O}_\ell$ use i_ℓ as their exec-index or write-index. Let Λ_ℓ be the subsequence of Λ that contains only invocation and response events of operation calls on the sequentially writable object $A[i_\ell]$ executed by operation calls in $\mathcal{O}_\ell$. Subsequence Λ_ℓ is a history, and by (4.1), no $A[i_\ell]$.`SWrite()` overlaps any other operation call on $A[i_\ell]$. Therefore, there is a linearization S_ℓ of Λ_ℓ. (Note that S_ℓ is the empty history if Λ contains no operation call M with $\texttt{TS}(M) = \ell$.)

Claim 4.7. *For any $\ell \in \mathbb{N}$ and any non-empty history Λ_ℓ, any linearization S_ℓ of Λ_ℓ starts with an* `SWrite()` *call and this is the only* `SWrite()` *in S_ℓ.*

Proof. Consider a non-empty history Λ_ℓ. Let p be the process that owns i_ℓ, and let S_ℓ be a linearization of Λ_ℓ. Suppose for the sake of contradiction, that there are two distinct $A[i_\ell]$.`SWrite()` operation calls SW_1 and SW_2 in S_ℓ. Let W_1 and W_2 be the `Write()` operation calls in which SW_1, respectively, SW_2 are executed. Since only the owner of i_ℓ can

execute an $A[i_\ell]$.`SWrite()` operation call, W_1 and W_2 are both executed by process p. So we can assume w.l.o.g. that W_1 happens before W_2. By the definition of Λ_ℓ, operation calls W_1 and W_2 are in $\mathcal{O}_\ell$, and therefore, $\texttt{TS}(W_1) = \texttt{TS}(W_2) = \ell$. By (4.3), this implies that $\texttt{TS}_{Ptr}(t(W_1)) = \texttt{TS}_{Ptr}(t(W_2))$. However, $\texttt{TS}_{Ptr}(t)$ denotes the number of writes to Ptr that happen in the interval $[0,t]$, and $t(W_1)$ and $t(W_2)$ are distinct points immediately after such writes. This implies that $\texttt{TS}_{Ptr}(t(W_1)) < \texttt{TS}_{Ptr}(t(W_2))$. This is a contradiction as both W_1 and W_2 have the same timestamp value. Therefore, there is at most one $A[i_\ell]$.`SWrite()` operation call in S_ℓ.

Next, to conclude the proof, we show that any `OP()` call Op in S_ℓ must be preceded by an `SWrite()` in S_ℓ. Since Λ_ℓ contains only operation calls on $A[i_\ell]$, Op also operates on the same object. Moreover, Op must have been executed by some process q during an `Exec()` operation call E. Operation call $Op \in \Lambda_\ell$, and so by the definition of Λ_ℓ, we know that $E \in \mathcal{O}_\ell$, so $\texttt{TS}(E) = \ell$.

Since $t(E)$ is the latest point between the invocation of E and the invocation of Op, such that (C1) is satisfied for $t^* = t(E)$, we have $Ptr = i_\ell$ at $t(E) \in [t_{E@inv}, t_{Op@inv}]$. This implies that some process p (possibly $p = q$) writes index i_ℓ to Ptr at some point before $t(E)$ during some `Write()` operation call W, because Ptr is initially set to $\perp$. Hence, $Ptr = i_\ell$ throughout $[t(W), t(E)]$, and therefore $\texttt{TS}_{Ptr}(t(W)) = \texttt{TS}_{Ptr}(t(E))$. We showed $\texttt{TS}(E) = \ell$, thus by (4.3) we have

$$\texttt{TS}(W) = \texttt{TS}(E) = \ell \tag{4.4}$$

By the template, process p completes an $A[i_\ell]$.`SWrite()` operation call SW before it writes i_ℓ to Ptr during W, and so before $t(W)$. Since $t(W) < t(E)$ and $t(E) \in [t_{E@inv}, t_{Op@inv}]$, p completes SW before the invocation of Op. By (4.4), we have $W \in \mathcal{O}_\ell$, and so $SW \in \Lambda_\ell$. Hence, SW linearizes before Op in Λ_ℓ, and therefore SW appears before Op in S_ℓ. □

We construct a sequential history S from Λ as follows. Consider a timestamp value $\ell \in \mathbb{N}$, and let k_ℓ be the number of operation calls in S_ℓ. If $k_\ell \geq 1$, then by Claim 4.7,

$S_\ell = SW_\ell, Op_{\ell,1}, ..., Op_{\ell,k_\ell-1}$, where SW_ℓ is an $A[i_\ell]$.`SWrite()` operation call and $Op_{\ell,j}$ is the j-th $A[i_\ell]$.`Oper()` operation call in S_ℓ, for $j \in \{1,\ldots,k_\ell - 1\}$.

Let W_ℓ be the invocation event of the `Write()` call in which SW_ℓ is invoked, and a matching response event. Also let $E_{\ell,j}$ be the invocation event of the `Exec()` operation call in which $Op_{\ell,j}$ is invoked, and a matching response that has the same return value as $Op_{\ell,j}$. Then we define

$$S = W_1, E_{1,1}, E_{1,2}, \ldots, W_2, E_{2,1}, E_{2,2}, \ldots, W_3, E_{3,1}, E_{3,2}, \ldots \tag{4.5}$$

Operation calls SW_ℓ and $Op_{\ell,j}$ are in S_ℓ, which is a linearization of Λ_ℓ. Hence, W_ℓ and $E_{\ell,k}$ are matured operation calls with their invocation events in $\mathcal{O}_\ell$. Therefore, $\texttt{TS}(W_\ell) = \texttt{TS}(E_{\ell,k}) = \ell$.

In the following, we prove that S is a linearization of a completion of $\Gamma(\Lambda)$. First we show that S contains all completed operation calls in $\Gamma(\Lambda)$. Consider any operation call M that completes in $\Gamma(\Lambda)$, and so it completes in Λ. Then M is a matured operation call, and there is some value $\ell \in \mathbb{N}$, where $\texttt{TS}(M) = \ell$. Hence, $M \in \mathcal{O}_\ell$, and by Claim 4.6, all operation calls in $\mathcal{O}_\ell$ invoke an operation on the same object $A[i_\ell]$, for some i_ℓ. Let Λ_ℓ be the subsequence of events in Λ that contains only invocation and response events on object $A[i_\ell]$ executed during operation calls in $\mathcal{O}_\ell$. Since M completes in Λ, its operation call Op on object $A[i_\ell]$ also completes in Λ and therefore in Λ_ℓ. Thus, Op appears in the linearization S_ℓ of Λ_ℓ. Thus by the construction of S, M appears in S.

Now, we prove the validity of S. If we append any valid sequential history on $\mathcal{WT}$ that starts with a `Write()` operation to a valid sequential history on the same object, then we obtain a valid sequential history, because a `Write()` call changes the state the object. So it is enough to show that subsequence $W_\ell, E_{\ell,1}, E_{\ell,2}, \ldots, E_{\ell,k_\ell-1}$ of S, for any $\ell \in \mathbb{N}$, is valid. By the construction of S, the order of operation calls $W_\ell, E_{\ell,1}, E_{\ell,2}, \ldots$ in S is the same as the order of their corresponding operation calls $SW_\ell, Op_{\ell,1}, Op_{\ell,2}, ...$ on $A[i_\ell]$ in S_ℓ. Sequential history S_ℓ is a valid sequential history, and the return value of each $E_{\ell,j}$ operation call is the same as the return value of its corresponding operation call $Op_{\ell,j}$ on $A[i]$, for any $j \in \{1,\ldots,k_\ell - 1\}$. Therefore, subsequence $W_\ell, E_{\ell,1}, E_{\ell,2}, \ldots$ is valid, and so S is valid.

To prove that the happens before order of operations is preserved in S, consider two operation calls M_1 and M_2 in $\Gamma(\Lambda)$, such that M_1 happens before M_2 in $\Gamma(\Lambda)$. We prove that M_1 precedes M_2 in S. Both M_1 and M_2 are matured. First suppose $\texttt{TS}(M_1) = \texttt{TS}(M_2) = \ell$. By the construction of S, each of M_1 and M_2 invokes an operation on the same sequentially writable object. Let Op_1 and Op_2 be the operations invoked by M_1 and M_2, respectively. Operation calls M_1 and M_2 both have the same timestamp value ℓ, and they both appear in S. Hence both Op_1 and Op_2 appear in S_ℓ, and therefore in Λ_ℓ. As M_1 happens before M_2 in $\Gamma(\Lambda)$, and so in Λ, operation Op_1 also happens before Op_2 in Λ_ℓ, and therefore Op_1 precedes Op_2 in S_ℓ. Thus by the construction of S, M_1 precedes M_2 in S.

Now suppose $\texttt{TS}(M_1) \neq \texttt{TS}(M_2)$. By (4.3), we have $\texttt{TS}(M_1) = \texttt{TS}_{Ptr}(t(M_1))$ and $\texttt{TS}(M_2) = \texttt{TS}_{Ptr}(t(M_2))$, where $t(M_1) \in [t_{M_1@inv}, t_{M_1@rsp}]$ and $t(M_2) \in [t_{M_2@inv}, t_{M_2@rsp}]$. Since M_1 completes before M_2 is invoked, $t(M_1) < t(M_2)$. Thus by (4.2) and since $\texttt{TS}(M_1) \neq \texttt{TS}(M_2)$, we have $\texttt{TS}_{Ptr}(t(M_1)) = \texttt{TS}(M_1) < \texttt{TS}(M_2) = \texttt{TS}_{Ptr}(t(M_2))$. Therefore, by the construction of S, M_1 precedes M_2 in S.

4.4.2 Correctness of the Proposed Implementation

In this section, we prove that our proposed wait-free implementation $\mathcal{WT}$ from instances of sequentially writable object $\mathcal{SW}$ of the same type $\mathcal{T}$ is linearizable. For that, we show in Lemma 4.8 that our algorithm of Figure 4.5 satisfies the conditions of Lemma 4.5.

Notation and Definitions. Suppose process q executes $A[i]$.`Oper()` in line 43 during an `Exec()` operation call E, for some exec-index i. We say q obtains exec-index i *directly* if q does not execute line 42 during E, and otherwise it obtains i *indirectly*.

In this pseudocode, we employ two helper operations `recycle()` and `updateQ()`. Since they are only helper functions that can only be called from other operation calls on the object, their invocations and responses do not appear in the transcript. During a `recycle()` operation, the calling process executes several operation calls on atomic registers, as well as local variables. For

the ease of explanation and to be consistent to operation calls that appear in the transcript, we define invocation point $t_{R@inv}$, respectively response point $t_{R@rsp}$, for any `recycle()` operation call R, to be the points when the first, respectively last, atomic operation call during R is executed. Similar to operation calls on objects, we say a `recycle()` call completes in an interval $[t_1, t_2]$ if $[t_{R@inv}, t_{R@rsp}] \subseteq [t_1, t_2]$. Operation `updateQ()` can be called in a `recycle()` operation R, and contains only steps on local variables. So all its local steps are executed at the same point as the latest preceding atomic operation call executed in R if there exists any, or at point $t_{R@inv}$ otherwise.

Transcript Λ. Suppose Λ' is an arbitrary transcript on object $\mathcal{WT}$ generated by executing `Write()` and `Exec()` operation calls starting from the initial state. We consider instead a transcript Λ that is obtained by prefixing Λ' with a `Write(`v_0`)` by process 0 on object $\mathcal{WT}$ as initialized in Figure 4.5. Since the initial state of $\mathcal{WT}$ is obtained by such a `Write(`v_0`)`, in order to prove linearizability of $\Gamma(\Lambda')$, it is enough to prove that $\Gamma(\Lambda)$ is linearizable.

Lemma 4.8. *Suppose some process q executes an* `Exec()` *operation call E in Λ, in which q invokes an $A[i^*]$.*`Oper()` *operation Op, for some i^*. Then there exists a point $t^* \in [t_{E@inv}, t_{Op@inv}]$, such that:*

(a) $Ptr = i^*$ at t^*, and

(b) no $A[i^*]$.`SWrite()` *operation call overlaps with interval $[t^*, t_{Op@rsp}]$.*

In the remainder of this section, we prove this lemma for transcript Λ. This together with Lemma 4.5 proves that our implementation is linearizable. To that end, we have to rely on several claims that describe properties and invariants that hold in Λ.

All about Recycling In the following, we discuss properties of the helper operation `recycle()`, and the local variables that it maintains. The first claim states that a process p can only store indices it owns in its queues or in its set $free_p$.

Claim 4.9. *At any point and for any process p, the following statements are true.*

(a) All values in $annc\mathcal{Q}_p$, $used\mathcal{Q}_p$, $hint\mathcal{Q}$, and $hint'\mathcal{Q}_p$ are either $\perp$ or indices owned by p,

(b) any value in $free_p$ and any value returned from a `recycle()` *call by p is an index owned by p, and*

(c) p only writes indices that it owns to Ptr.

Proof. We prove each part separately.

Proof of Part (a). This statement is true at the beginning of any execution, because all queues contain only $\perp$ values. Process p can add an element to any of it queues only in lines 49 and 51. In these two lines, p adds an index that it owns (see line 47), or a $\perp$ value. Hence, the claim of Part (a) is true.

Proof of Part (b). Since initially $free_p$ contains all indices that p owns and no `recycle()` call has been executed, the claim of Part (b) is true at the beginning of any execution. Process p can add an element to $free_p$ only in line 55, and it only adds an index that it removed from one of its queues. By Part (a), p owns all (non-$\perp$) indices that are stored in its queues. Thus, p only adds indices that it owns to $free_p$, and so any value in set $free_p$ is owned by p. Moreover, p's `recycle()` call returns a value that is removed from $free_p$, and since every value stored in $free_p$ is owned by p, the claim holds.

Proof of Part (c). Process p can write an index f to Ptr only in line 20 or 23 during a `Write()` operation call. Either of these can happen only if p's latest preceding `recycle()` operation call in line 12 returns f. By Part (b), any value returned from a `recycle()` call by p, and in particular index f, must be an index owned by p. Therefore, the claim of Part (c) is true. □

By the result of Claim 4.9, we get the following Corollary.

Corollary 4.10. *For any process p and any index $i^* \in I_p$, only p can execute an $A[i^*]$.*`SWrite()` *operation, and thus no two $A[i^*]$.*`SWrite()` *operation calls can overlap.*

Proof. A process only calls $A[i^*]$.`SWrite()`, after a `recycle()` call that returned i^*. Hence by Claim 4.9(b), only process p can execute an $A[i^*]$.`SWrite()` operation. Since processes are sequential, no two $A[i^*]$.`SWrite()` operation calls can overlap. □

The following three claims establish some invariants about the queues and array c_p. First, we show that at each point in time all queues have size $2n+2$, and the value stored in $c_p[i]$ is the total number of copies of index i in those queues. We also show that every index i that is owned by p is stored either in one of p's queues or in $free_p$.

Claim 4.11. *At any point, each of the queues $usedQ_p$, $anncQ_p$, $hintQ_p$, and $hint'Q_p$ has size exactly $2n+2$.*

Proof. Each of p's queues initially has $2n+2$ elements with value $\perp$. These queues can only be modified when p calls `updateQ()` during a `recycle()` operation call R in one of lines 26, 28, 30 and 32. The `updateQ()` operation contains no operation calls on any shared objects, and so it is executed at the same point as the latest preceding atomic operation call, or with the invocation of R if there is no preceding atomic call in R. Moreover, during this operation call, process p enqueues exactly one element into one of its queues (line 49 or 51), and dequeues exactly one element from the same queue (line 52). Thus, an `updateQ()` operation call by p does not change the size of any of p's queues. □

Claim 4.12. *For any process p, and any index $i \in I_p$, at any point, the value of $c_p[i]$ is the total number of occurrences of index i in $usedQ_p$, $anncQ_p$, $hintQ_p$, and $hint'Q_p$.*

Proof. Fix some process p and an index $i \in I_p$. Initially, all of the queues contain only $\perp$ elements, and $c_p[i] = 0$, so the claim is true. Process p's queues and the value stored in $c_p[i]$ can be modified only during an `updateQ()` operation call by p, and all those steps are mapped to the same point. During this operation call, if p enqueues an index i into one of its queues, then and only then, it increments the value of $c_p[i]$ (lines 48–49). Similarly, if p dequeues an

index i from one of its queues, then and only then, it decrements the value of $c_p[i]$ (lines 52–54). Hence, the claim holds. □

Claim 4.13. *Consider some process p, and any index $i^* \in I_p$. At any point, $i^* \in usedQ_p \cup anncQ_p \cup hintQ_p \cup hint'Q_p \cup free_p$.*

Proof. Initially, all indices of I_p are in $free_p$, and so $i^* \in free_p$ at the beginning of the execution. Only p can access p's queues and $free_p$. Suppose process p removes i^* from $free_p$ at some point t. This can only happen in line 25 of a `recycle()` call. At the same point, p executes its `updateQ(`$usedQ_p$,i^*`)` operation call in line 26. Since process p owns index i^*, the if-condition in line 47 of this operation call evaluates to true, and so p enqueues i^* into $usedQ_p$ in line 49. All these steps are on p's local variables, and so they all happen at t. Thus, an instance of i^* is in p's queues right after this point.

Next suppose that process p removes the last instance of i^* from its queues at some point t. This can only happen in line 52 of an `updateQ()` operation call by p. By Claim 4.12, $c_p[i^*] = 0$ when p executes lines 52–55. Thus p adds i^* to $free_p$ in line 55. Since all these steps are on p's local variables, i^* is in $free_p$ right after t. □

In the following claim, we show that there is always at least one index in $free_p$, when p executes $free_p$`.remove()` in line 25 of its `recycle()` call.

Claim 4.14. *For any process p, $|free_p| \geq 1$ just before p executes line 25 of its* `recycle()` *operation call.*

Proof. Fix a process p, a `recycle()` operation call R by p, and let t be the point just before p executes line 25 of R. By Claim 4.11, each of p's queues, $usedQ_p$, $anncQ_p$, $hintQ_p$, and $hint'q_p$ contains $2n+2$ elements at t. Thus, at this point there can be at most $8n+8$ distinct indices of I_p in p's queues. By Claim 4.13, each index $i^* \in I_p$ is in $free_p$ or in one of p's queues at t. Since $|I_p| = 8n+9$, there is at least one index of I_p that is not in any of the queues, an therefore in $free_p$ at this point. □

In the following three claims, we discuss how and when an index gets added to one of p's queues or $free_p$. More specifically in Claim 4.15, we show that once an index is added to one of p's queues, then it remains in that queue, and therefore, it cannot be added to $free_p$, at least until p has completed another $2n+1$ `recycle()` calls. Moreover in Claim 4.16, we show that when some process p reads some index i from an entry of $Annc$, H_p, or H'_p during a `recycle()` call, then starting from that point, p only can add i to $free_p$ after it completes at least $2n+1$ more `recycle()` calls, and after i is not anymore stored in $Annc$, H_p, and H'_p, whichever is last. In Claim 4.17 we use this to show that if p executes at least n `recycle()` calls during some interval and one of $Annc$, H_p, and H'_p holds some value i throughout the same interval, then p reads index i from that register at some point during this interval and keeps it out of $free_p$ until the end of this interval.

Claim 4.15. *Consider a* `recycle()` *operation call R by some process p. Let $R_1, R_2, \ldots$ be the* `recycle()` *calls that p executes after R, and let $t' = t_{R_{2n+2}@inv}$ if p invokes R_{2n+2} in Λ, and $t' = \infty$ otherwise. Also let qu be one of p's queues $usedQ_p$, $anncQ_p$, $hintQ_p$, or $hint'Q_p$. If p enqueues $i^* \in I_p$ into qu at some point t during R, then*

(a) that instance of i^* does not get removed from qu throughout $[t, t')$, and

(b) process p does not add i^* to $free_p$ throughout $[t, t')$.

Proof. By Claim 4.11, the size of each of p's queues is $2n+2$. During each `recycle()` operation call, p dequeues exactly one element from each of these queues (in lines 26, 28, 30 and 32). Therefore, when p enqueues index i^* into qu during R, there are $2n+2$ elements in front of that instance of i^* in the queue. One element is dequeued during R, and the remaining $2n+1$ elements get dequeued during $R_1, \ldots, R_{2n+1}$. Thus, after an instance of i^* is enqueued to qu at t, this instance of i^* gets removed from qu during R_{2n+2}, if p invokes $2n+2$ additional `recycle()` calls after t, and otherwise it remains in qu until the end of Λ. This proves Part (a).

By Part (a), an instance of i^* is in one of p's queues throughout $[t, t')$, and so by Claim 4.12, we have $c_p[i^*] \geq 1$, throughout this interval. Since p can only add index i^* to $free_p$ in line 55

when $c_p[i^*] = 0$, process p does not add i^* to $free_p$ throughout $[t,t')$, so Part (b) follows. $\square$

Claim 4.16. *Consider an index $i^* \in I_p$, and a* `recycle()` *operation call R by some process p. Let X denote either $Annc[q]$, the first component of $H_p[q]$, or the first component of $H'_p[q]$. Suppose that at some point t during R, process p reads i^* from X. Let $R_1, R_2, \ldots$ be the* `recycle()` *calls that p executes after R, and let*

(a) $t' = t_{R_{2n+2}@inv}$, if p invokes R_{2n+2} in Λ, and $t' = \infty$ otherwise, and

(b) t'' be the first point after t at which the value of X is changed, and $t'' = \infty$ if such a point does not exist in Λ.

Then p does not add i^* to $free_p$ throughout $[t, \max\{t', t''\})$.

Proof. Suppose process p reads i^* from X at point t during R. This happens either in line 27, 29, or 31 of R, depending on whether X is $Annc[q]$, or the first component of $H_p[q]$ or of $H'_p[q]$. In any of these cases, process p adds i^* to one of its queues at the same point t as it calls `updateQ(·,i*)`. By Claim 4.15 (b), p does not add i^* to $free_p$ throughout $[t,t')$. So if $t'' \leq t'$, the claim follows.

Now suppose that $t'' > t'$. We show that p does not add i^* to $free_p$ throughout $[t',t'')$. For the purpose of contradiction, assume p adds i^* to $free_p$ at some point $t_2 \in [t',t'')$, and let t_1 be the last point during $[t,t_2]$ at which p reads i^* from X, during some `recycle()` operation call. Since this happens at point t, such a point t_1 exists in the interval $[t,t_2]$. By the claim assumption in Part (b), $X = i^*$ throughout $[t,t'')$, and thus

$$X = i^* \text{ throughout } [t_1,t_2]. \tag{4.6}$$

During each `Write()` call, and thus once between every two `recycle()` calls, p increments modulo n the value stored in ctr_p, and it does not modify ctr_p anywhere else. Therefore,

$$\text{process } p \text{ must complete fewer than } n \texttt{ recycle()} \text{ calls during } [t_1,t_2], \tag{4.7}$$

since otherwise p executes at least one complete `recycle()` call R^* during $[t_1, t_2]$, such that $ctr_p = q$ at the invocation of R^*. Thus, by (4.6), p reads i^* from X in R^*. This contradicts the assumption that at point t_1, process p reads i^* from X for the last time before t_2, and so (4.7) is true.

At t_1 process p reads i^* from X, and then adds i^* to one of its queues, (in either lines 27–28, lines 29–30, or lines 31–32, depending on whether X is $Annc[q]$, or the first component of $H_p[q]$ or of $H'_p[q]$). Thus by Claim 4.15 (b), p does not add i^* to $free_p$ after t_1 until it completes another $2n+1$ `recycle()` calls, which is by (4.7) some time after t_2. This is a contradiction, because we assumed that p adds i^* to $free_p$ at t_2. □

Claim 4.17. *Consider some processes p and q, and let X denote either $Annc[q]$, or the first component of $H_p[q]$, or the first component of $H'_p[q]$. Suppose $X = i^*$ throughout some interval $[t_1, t_2]$. If p executes at least n `recycle()` calls $R_1, \ldots, R_n$ during $[t_1, t_2]$, then p does not add i^* to $free_p$ throughout $[t_{R_n@rsp}, t_2]$.*

Proof. Process p increments ctr_p modulo n once during each `Write()` operation call and thus once between each two `recycle()` calls. Therefore, p completes at least one `recycle()` call R_k, $1 \leq k \leq n$, during $[t_1, t_2]$, such that $ctr_p = q$ at the invocation of R_k. Thus process p reads i^* from X in line 27, line 29, or line 31 of R_k, depending on what X denotes. Then at the same point during R_k, process p enqueues i^* to one of its queues at some point $t^* \in [t_1, t_{R_k@rsp}]$. Therefore, by Claim 4.16, index i^* does not get added to $free_p$ during the interval that starts at t^* and ends when the value stored in X changes for the first time after t^*. Thus, p does not add i^* to $free_p$ throughout $[t^*, t_2]$ and therefore p does not add i^* to $free_p$ throughout $[t_{R_n@rsp}, t_2]$, because $t^* < t_{R_k@rsp} < t_{R_n@rsp}$. □

Hint Mechanism Here we provide some claims about hints a process p provides to another process q. In particular in Claim 4.18, we show that if the sequence number attached to the latest hint provided to q by p is the same as the current value of S_q, then p does not provide a new hint, since q might be using the provided hint as its exec-index.

Next, in Claim 4.19, we show that at each point when some process p is not in the process of providing a hint, either $H_p[q]$ or $H'_p[q]$ stores $(\perp,\perp)$, or the first components of $H_p[q]$ and $H'_p[q]$ are equal, for any q.

Claim 4.18. *Consider two processes p and q. Let t be a point when p has either no pending* `Write()` *call, or its* `Write()` *call W is pending, but $t \notin [t_{W@14}, t_{W@21})$. Also let t' be the first point after t, at which q writes to one of $H_p[q]$, $H'_p[q]$, and S_q, and $t' = \infty$ if such a point does not exist. Suppose at point t, we have $H_p[q] = (j_1, s^*)$, $H'_p[q] = (j_2, s^*)$, and $S_q = s^*$, for some values j_1, j_2, s^*. Then process p writes to neither $H_p[q]$ nor $H'_p[q]$ throughout (t, t').*

Proof. Suppose for the sake of contradiction that p writes to at least one of $H_p[q]$ and $H'_p[q]$ during (t, t'). Let $X \in \left\{H_p[q], H'_p[q]\right\}$ be the register to which p writes first during this interval, and let W' be the `Write()` call by p in which this happens. Then $ctr_p = q$ when p invokes W', and p writes to X at some point t^*, where $t^* = t_{W'@18}$ if $X = H_p[q]$, and $t^* = t_{W'@21}$ if $X = H'_p[q]$. So $t^* \in (t, t')$. We have $t < t^*$ and by the claim assumption, $t \notin [t_{W'@14}, t_{W'@21})$. Thus, $t < t_{W'@14}$.

Recall that only q can write to S_q, and only p and q can write to $H_p[q]$ and $H'_p[q]$. By the claim assumption, $S_q = s^*$ throughout $[t, t')$, process q does not write to any of $H_p[q]$ and $H'_p[q]$ during (t, t'), and t^* is the first point during this interval at which p writes to any of these variables. Hence, $S_q = s^*$, $H_p[q] = (j_1, s^*)$, $H'_p[q] = (j_2, s^*)$ throughout (t, t^*) and therefore throughout $[t_{W'@14}, t^*)$ (because $t < t_{W'@14}$). This implies that process p reads the same sequence number s^* from S_q at $t_{W'@14}$, and from the second component of $H_p[q]$ at $t_{W'@15}$, and from the second component of $H'_p[q]$ at $t_{W'@16}$. Thus, the if-condition in line 17 of W' evaluates to false, and p does not execute lines 18–21 of W', which is a contradiction. □

Claim 4.19. *Consider two processes p and q. Let t be a point when p has either no pending* `Write()` *call, or its* `Write()` *call W is pending, but $t \notin [t_{W@18}, t_{W@21}]$. Then at t, either $H_p[q] = (\perp,\perp)$, or $H'_p[q] = (\perp,\perp)$, or the first components of $H_p[q]$ and $H'_p[q]$ are equal.*

Proof. Fix two processes p and q. Initially, we have $H_p[q] = H'_p[q] = (\perp, \perp)$, and so the claim holds. Recall that only processes p and q can write to $H_p[q]$ and $H'_p[q]$. Process p only writes to one of these two registers in lines 18 and 21, respectively, of a `Write()` operation call. By the `Write()` implementation, if p writes some pair (j, s) to one of $H_p[q]$ and $H'_p[q]$ during a `Write()` call, it writes a pair (j, s') to the other one during the same `Write()` call, where s and s' may or may not be equal. Process q only writes $(\perp, \perp)$ to one of these two registers in lines 44 and 45, respectively, of its `Exec()` operation call. Thus, if p is not poised to execute lines 18–21, there are two possible cases: Either the last writes to $H_p[q]$ and $H'_p[q]$ were both by process p, and thus the first components of $H_p[q]$ and $H'_p[q]$ are equal, or the last write to at least one of these two registers was by process q, and thus one of $H_p[q]$ and $H'_p[q]$ stores value $(\perp, \perp)$. □

The following claim uses the results of both Claims 4.18 and 4.19. Later in the proof of Lemma 4.8, we use this claim to show that if p completes at least n `Write()` operation calls between the point that some process q reads an index $i \in I_p$ from Ptr and the point right after q announces this index, then q's following if-condition in line 41 evaluates to true.

Claim 4.20. *Consider a* `Write()` *operation call W by some process p, and let q be the value of ctr_p at $t_{W@inv}$. Suppose E is an* `Exec()` *call by q, and W completes during $(t_{E@35}, t_{E@40}]$. Then at $t_{W@rsp}$, we have $H_p[q] = H'_p[q] = (j, s^*)$, for some j, where s^* is the value q writes to S_q at $t_{E@35}$.*

Proof. An illustration of this proof is depicted in Figure 4.6. Process q writes s^* to S_q at $t_{E@35}$, and it does not write to S_q during time interval $(t_{E@35}, t_{E@40}]$. Therefore, since only q writes to S_q, and it only writes non-$\perp$ values, we have

$$S_q = s^* \neq \perp \text{ throughout } [t_{W@inv}, t_{W@rsp}]. \tag{4.8}$$

Only processes p and q can write to $H_p[q]$ and $H'_p[q]$. Process q does not write to either of these two registers during time interval $(t_{E@35}, t_{E@40}]$, hence

$$q \text{ does not write to either } H_p[q] \text{ or } H'_p[q] \text{ throughout } [t_{W@inv}, t_{W@rsp}]. \tag{4.9}$$

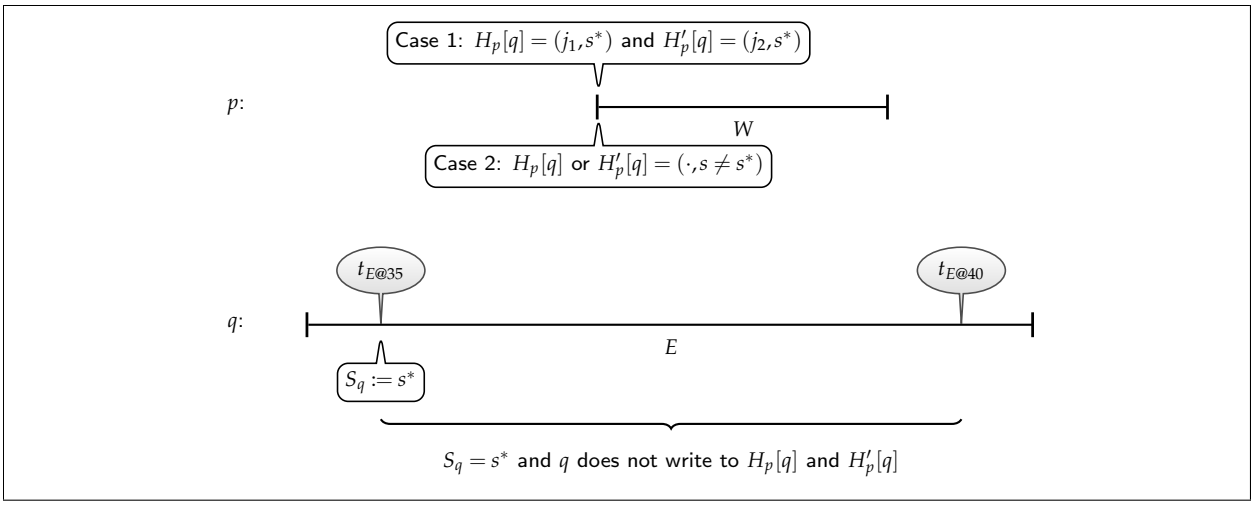

Figure 4.6: Illustration for the proof of Claim 4.20

Now we consider the following two cases: First, consider the case in which $H_p[q] = (j_1, s^*)$ and $H'_p[q] = (j_2, s^*)$ at $t_{W@inv}$, for some values j_1 and j_2. By (4.8), $s^* \neq \perp$, and therefore by Claim 4.19, j_1 must be equal to j_2. Moreover by (4.8), $S_q = s^*$ at $t_{W@inv}$. Hence, by Claim 4.18, p does not write to either $H_p[q]$ or $H'_p[q]$ between $t_{W@inv}$ and the point at which q writes to either S_q, $H_p[q]$, or $H'_p[q]$. Hence by (4.8) and (4.9), $H_p[q] = H'_q[q] = (j, s^*)$ at the response of W, where $j = j_1 = j_2$.

Next suppose that at $t_{W@inv}$ at least one of $H_p[q]$ and $H'_p[q]$ stores some pair $(\cdot, s)$, where $s \neq s^*$. Process p does not write to any of these two registers between the invocation and line 18 of W, and by (4.9), process q also does not write to them during the same interval. Hence, the second component stored in at least one of $H_p[q]$ and $H'_p[q]$ is $s \neq s^*$ when p reads it in line 15 or 16 of W. By (4.8), $S_q \neq s$ when q reads S_q in line 14, and thus the if-condition in line 17 evaluates to true. So, p writes pair (j, s^*) to $H_p[q]$ in line 18 and to $H'_p[q]$ in line 21 of W for some value j. Process p does not modify the value stored in these two registers anymore during W, and so $H_p[q] = H'_q[q] = (j, s^*)$ at $t_{W@rsp}$. □

Write Operation In the following section, we discuss all the invariants that hold for some object $A[i^*]$ that is put into use during a `Write()` operation call W by some process p. In

particular, in Claim 4.22, we prove that p completes at least $2n+2$ `recycle()` calls between the point at which $A[i^*]$ is in use and the next point at which p invokes an $A[i^*]$.`SWrite()` call. Moreover, p can only add i^* to $free_p$, after it completes $2n$ `recycle()` calls after the point at which $A[i^*]$ is in use.

To achieve that, we first prove that i^* is in $free_p$ just before W is invoked. Then we show that if p puts the same object $A[i^*]$ into use again during a later `Write()` operation call W', then p executes at least $2n+2$ `recycle()` calls between W and W' (see Claim 4.21 for more details).

Finally in Claim 4.23, we prove that while $A[i^*]$ is in use, no $A[i^*]$.`SWrite()` operation call is pending.

Claim 4.21. *Let W and W' be two distinct* `Write()` *operation calls by some process p, such that W is executed before W' in Λ. Also let R_W and $R_{W'}$ be the* `recycle()` *calls executed during W and W', respectively, and $R_1, R_2, \ldots$ be the* `recycle()` *calls p execute after R_W. Suppose $f = i^* \in I_p$ at both $t_{R_W@rsp}$ and $t_{R'_W@rsp}$, then*

(a) p completes at least $2n+2$ `recycle()` *calls during $[t_{W@rsp}, t_{W'@inv})$.*

(b) p adds i^* to $free_p$ at some point during $[t_{R_{2n+2}@inv}, t_{W'@inv})$,

Proof. Since $f = i^*$ at $t_{R_W@rsp}$ and $t_{R'_W@rsp}$, both operation calls R_W and $R_{W'}$ must have returned i^*. This implies that p removes i^* from $free_p$ in line 25 of R_W at $t_{R_W@inv} = t_{W@inv}$ and in line 25 of $R_{W'}$ at $t_{R_{W'}@inv} = t_{W'@inv}$. Therefore,

$$i^* \in free_p \text{ just before } t_{W@inv} \text{ and } t_{W'@inv}. \tag{4.10}$$

Proof of Part (a). Since R_W returns index i^*, this index is removed from $free_p$ and enqueued to $usedQ_p$ in lines 25–26 of R_W at $t_{R_W@inv} = t_{W@inv}$, as depicted in Figure 4.7. Let $t' = t_{R_{2n+2}@inv}$ if p invokes R_{2n+2} in Λ, and $t' = \infty$ otherwise. By Claim 4.15 (b), p does not add i^* to $free_p$ throughout $[t_{W@inv}, t')$. Therefore,

$$i^* \notin free_p \text{ throughout } [t_{W@inv}, t'). \tag{4.11}$$

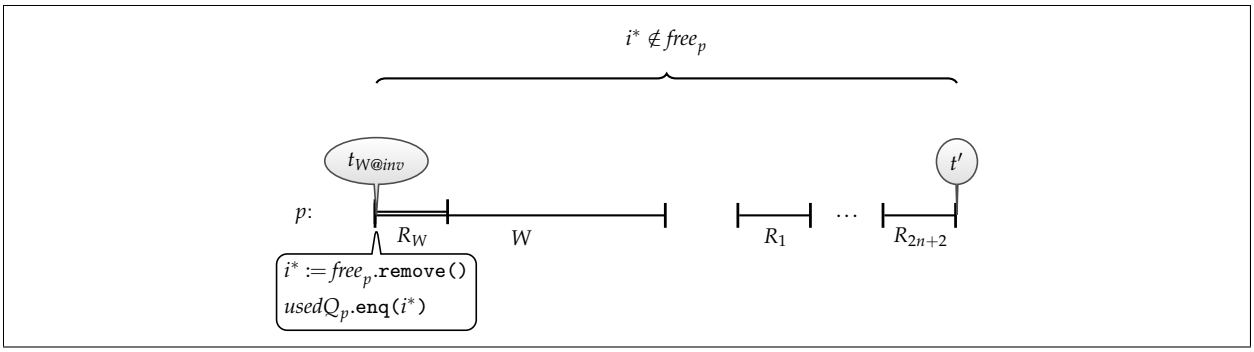

Figure 4.7: Illustration for the proof of Claim 4.21, Part (a)

By (4.10), index i^* is in $free_p$ just before $t_{W'@inv} > t_{W@inv}$. This in addition to (4.11) implies that $t' < t_{W'@inv} < \infty$, and so

$$t' = t_{R_{2n+2}@inv}. \tag{4.12}$$

Therefore, p invokes R_{2n+2} before it invokes W', and so it must complete this operation call before W' as well. Since p invokes only one `recycle()` call during each `Write()` operation call, R_1 is invoked after W responds. Therefore, at least all $2n+2$ `recycle()` calls $R_1,\ldots,R_{2n+2}$ complete during the interval that starts with the response of W and ends with the invocation of W'.

Proof of Part (b). By (4.11), (4.12), (4.10) and Part (a), index i^* is not in $free_p$ throughout $[t_{W@inv}, t_{R_{2n+2}@inv})$, but it is in $free_p$ just before $t_{W'@inv} > t_{R_{2n+2}@inv}$. Thus, index i^* must be added to $free_p$ at some point during $[t_{R_{2n+2}@inv}, t_{W'@inv})$, and by Claim 4.9 (b), this must be done by the owner of index i^*, process p. □

Claim 4.22. *Consider some process p. Suppose $Ptr = i^* \in I_p$ at some point t_1, and process p invokes an $A[i^*]$.`SWrite(x)` call at a later point $t_2 > t_1$. Then*

(a) p completes at least $2n+2$ `recycle()` calls during interval $[t_1, t_2]$, and

(b) Let R_{2n+1} be p's $2n+1$-st `recycle()` call after t_1, then p adds i^* to $free_p$ at some point during $[t_{R_{2n+1}@inv}, t_2)$.

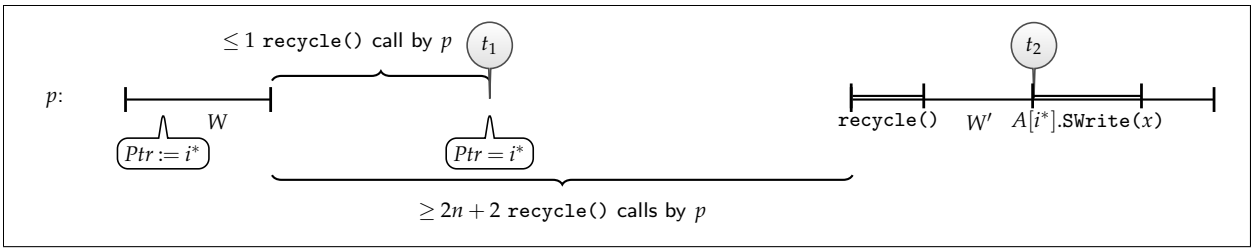

Figure 4.8: Illustration for the proof of Claim 4.22, Part (a)

Proof. At point t_1, index i^* is stored in Ptr as depicted in Figure 4.8. The value of Ptr can only change to i^* during a `Write()` operation call by process p (see Claim 4.9(c)). Let W be the `Write()` operation call during which p writes i^* into Ptr for the last time before t_1, and hence p's local variable f has value i^* during W. Let W' be the `Write()` operation call in which p invokes $A[i^*]$.`SWrite(`x`)` at point $t_2 > t_1$, and thus p's local variable f has value i^* during W', as well.

Process p invokes at most one `recycle()` call after W responds and before t_1, because otherwise if p invokes two `recycle()` calls during $[t_{W@rsp}, t_1]$, then this process writes to Ptr at least once during this interval, and since we assumed W is the last `Write()` operation call during which p writes i^* to Ptr before t_1, the value of Ptr would not be equal to i^* at t_1.

By Claim 4.21(a), process p completes at least $2n+2$ `recycle()` calls during $(t_{W@rsp}, t_{W'@inv})$. Since p executes at most one `recycle()` call during $(t_{W@rsp}, t_1)$,

$$p \text{ completes at least } 2n+1 \texttt{ recycle() } \text{calls during } [t_1, t_{W'@inv}), \tag{4.13}$$

By the implementation, process p completes another `recycle()` call during $[t_{R_{W'}@inv}, t_2]$. Hence by (4.13), p completes at least $2n+2$ `recycle()` calls during $[t_1, t_2]$. This completes the proof of Part (a).

Let R be the $2n+2$-nd `recycle()` call p invokes after $t_{W@rsp}$. By Claim 4.21(b), process p adds i^* to $free_p$ at some point during $[t_{R@inv}, t_{W'@inv}) \subseteq [t_{R@inv}, t_2)$ (because $t_{W'@inv} < t_2$). Since p invokes at most one `recycle()` call during $[t_{W@rsp}, t_1]$, operation call R is either the $2n+2$-nd or the $2n+1$-st `recycle()` call p invokes after t_1. Therefore, $t_{R@inv} \geq t_{R_{2n+1}@inv}$.

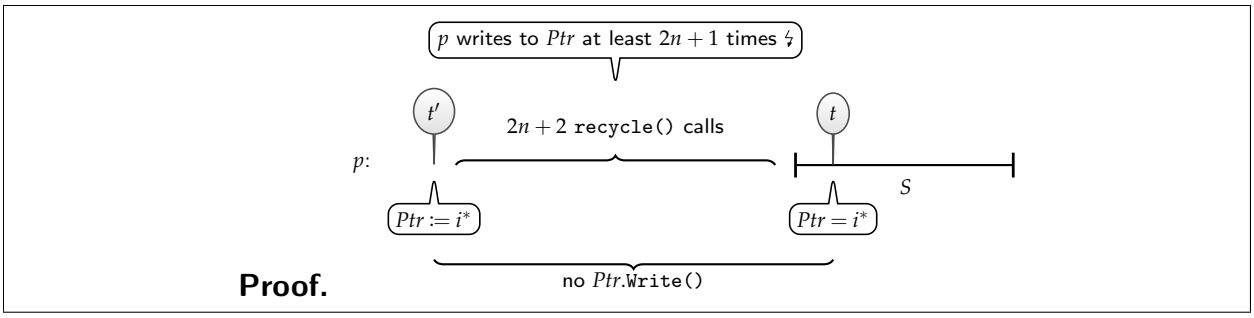

Figure 4.9: Illustration for the proof of Claim 4.23

Hence, p adds i^* to $free_p$ at some point during $[t_{R_{2n+1}@inv}, t_2)$, which proves Part (b). □

Claim 4.23. *Consider an index $i^* \in I_p$. If $Ptr = i^*$ at some point t, then no $A[i^*]$.`SWrite()` operation call is pending at t.*

By Claim 4.9 and Corollary 4.10, only process p can write i^* to Ptr and execute $A[i^*]$.`SWrite()`. Suppose, for the sake of contradiction, that p executes an $A[i^*]$.`SWrite()` operation call S that is pending at t as depicted in Figure 4.9. Since $Ptr = i^*$ at t, process p must have written i^* to Ptr at some point prior to t. Let t' be the last time before t at which p writes i^* to Ptr, thus

$$\text{no process writes to } Ptr \text{ during } (t', t]. \tag{4.14}$$

Since S gets invoked before t and is pending at t, and also p does not write to Ptr during an `SWrite()` operation call, point t' occurs before the invocation of S.

By Claim 4.22(a), p completes at least $2n + 2$ `recycle()` operation calls after t' and before the invocation of S. Hence, p completes at least $2n + 1$ `Write()` operation calls between t' and the invocation of S. Process p writes to Ptr exactly once during each `Write()` call (in line 20 or 23), so p writes to Ptr at least $2n + 1$ times after t' and before the invocation of S, and thus before t. This contradicts (4.14). □

Exec Operation In the following two claims, we establish the main properties needed to prove Lemma 4.8, when process q obtains an index indirectly during an `Exec()` call.

The next claim states that if during an `Exec()` call E, a process q reads identical hints from $H_p[q]$ and $H'_p[q]$ augmented with a sequence number s^* that matches q's current sequence number, then those hints are provided by p during one `Write()` call. To prove this, we assume for the purpose of contradiction that these hints are provided during two distinct `Write()` calls W and W'. We consider different cases of how W and W' are ordered compared to E, and we show how each case leads to a contradiction. In particular, we show that either one of these hints get overwritten before q reads it during E, or the same hint i^* cannot be provided during both of these operations, because i^* is continuously stored in an entry of H or H', and so it cannot be added to *Free* to be chosen as a write-index again.

Claim 4.24. *Consider an index $i^* \in I_p$ and an* `Exec()` *operation E by some process q. Suppose q reads (i^*,s^*) from both $H_p[q]$ and $H'_p[q]$ in line 39 respectively line 40 of E, where s^* is the value q writes to S_q in line 35 of E. Then there exists a* `Write()` *operation call by p during which p writes (i^*,s^*) into $H_p[q]$ and $H'_p[q]$ both for the last time before $t_{E@40}$.*

Proof. Process q reads (i^*,s^*) from both $H_p[q]$ and $H'_p[q]$ at $t_{E@39}$ and $t_{E@40}$, respectively. Therefore, process p must have previously written (i^*,s^*) to $H_p[q]$ and $H'_p[q]$ during some `Write()` call(s) in lines 18 and 21, respectively. We define W and W' as follows:

$$\begin{aligned}&W \text{ is the last } \texttt{Write()} \text{ call by } p\text{, such that } t_{W@18} < t_{E@40} \text{ and}\\&\text{at } t_{W@18}\text{, } p \text{ writes } (i^*,s^*) \text{ to } H_p[q].\end{aligned} \tag{4.15}$$

$$\begin{aligned}&W' \text{ is the last } \texttt{Write()} \text{ call by } p\text{, such that } t_{W'@21} < t_{E@40} \text{ and}\\&\text{at } t_{W'@21}\text{, } p \text{ writes } (i^*,s^*) \text{ to } H'_p[q].\end{aligned} \tag{4.16}$$

Process q reads (i^*,s^*) from $H'_p[q]$ at $t_{E@40}$, and since only p can write a non-$(\perp,\perp)$ pair to $H'_p[q]$,

$$H'_p[q] = (i^*,s^*) \text{ throughout } [t_{W'@21}, t_{E@40}]. \tag{4.17}$$

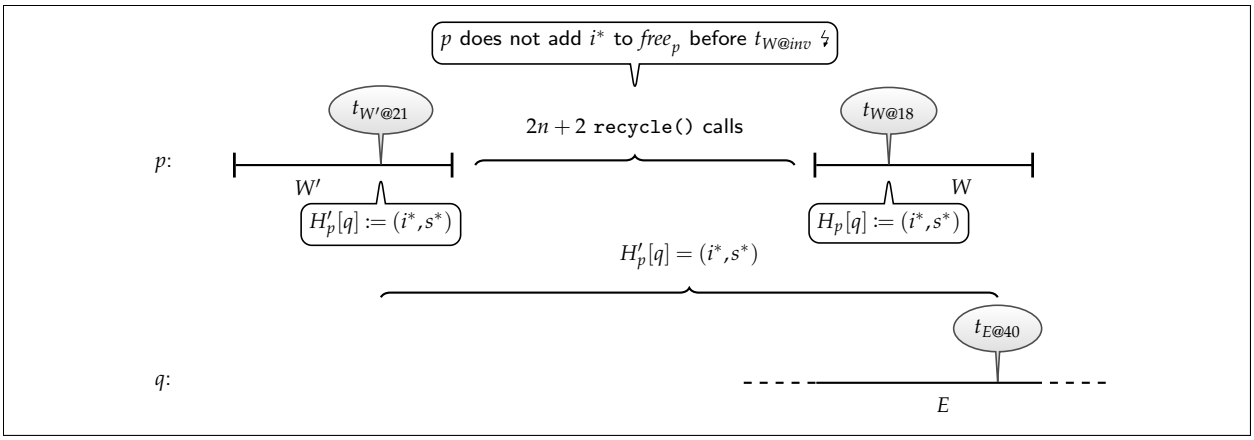

Figure 4.10: Illustration for the proof of (4.19) of Claim 4.25, Case 1

Now we show that

$$\text{if } t_{W@18} < t_{E@39}, \text{ then no process writes to } H_p[q] \text{ throughout } (t_{W@18}, t_{E@39}]. \tag{4.18}$$

For the sake of contradiction, suppose one or more writes to $H_p[q]$ occurs during $(t_{W@18}, t_{E@39}]$. The last of these writes must be a `Write`(i^*,s^*) call by p, because q reads (i^*,s^*) from $H_p[q]$ at $t_{E@39}$. Thus p writes (i^*,s^*) to $H_p[q]$ at some point during $(t_{W@18}, t_{E@39}] \subseteq (t_{W@18}, t_{E@40}]$, which contradicts (4.15).

To prove the claim, it is enough to show that

$$W = W'. \tag{4.19}$$

For the sake of contradiction, assume $W \neq W'$. Operation calls W and W' are both executed by process p, and so they are not overlapping. Therefore W gets executed either before W' or after W'.

Case 1. First suppose p executes W after W' as shown in Figure 4.10. This together with (4.15) implies $t_{W'@21} < t_{W@18} < t_{E@40}$. Thus, $[t_{W'@rsp}, t_{W@inv}] \subseteq [t_{W'@21}, t_{E@40}]$, and hence by (4.17) we obtain

$$H'_p[q] = (i^*,s^*) \text{ throughout } [t_{W'@rsp}, t_{W@inv}]. \tag{4.20}$$

By (4.15) and (4.16), p's local variable f has the same value i^* during both W and W', thus, Claim 4.21 applies. By Claim 4.21(a), p completes at least $2n+2$ `recycle()` calls $R_1,\ldots,R_{2n+2}$ during $[t_{W'@rsp}, t_{W@inv}]$. Therefore by Claim 4.17 and (4.20), p does not add i^* to $free_p$ throughout $[t_{R_n@rsp}, t_{W@inv}]$. This is a contradiction, because by Claim 4.21(b), process p adds i^* to $free_p$ at some point during $[t_{R_{2n+2}@inv}, t_{W@inv}) \subseteq [t_{R_n@rsp}, t_{W@inv}]$.

Case 2. Now suppose p executes W before W'. By (4.16) we have

$$t_{W@18} < t_{W'@18} < t_{W'@21} < t_{E@40}. \tag{4.21}$$

We consider the following cases:

Case 2.1: $t_{W'@18} < t_{E@39}$**.** Then by (4.21), $t_{W@18} < t_{E@39}$. Thus, by (4.18), no process writes to $H_p[q] = (i^*, s^*)$ throughout $(t_{W@18}, t_{E@39}]$. This is a contradiction, because by (4.16), process p provides a hint to q during W' and so p writes some pair to $H_p[q]$ at $t_{W'@18} \in (t_{W@18}, t_{E@39}]$.

Case 2.2: $t_{E@39} < t_{W'@18}$**.** First suppose $t_{E@35} < t_{W'@14}$. By (4.16), p writes (i^*, s^*) to $H'_p[q]$ at point $t_{W'@21}$. Thus, at $t_{W'@18}$, process p writes a pair (i^*, s) to $H_p[q]$, where s is the value p read from S_q at $t_{W'@14}$. Recall that we assumed $t_{E@35} < t_{W'@14}$, and so from (4.21), we obtain $t_{W'@14} \in (t_{E@35}, t_{E@40})$. Thus, we have $s = s^*$, i.e. p writes (i^*, s^*) to $H_p[q]$ at $t_{W'@18}$. According to (4.21), $t_{W'@18} \in (t_{W@18}, t_{E@40})$. This contradicts (4.15).

Now suppose $t_{W'@14} < t_{E@35}$ as shown in Figure 4.11. By the assumption that W gets executed before W', we have $t_{W@18} < t_{W'@14} < t_{E@35} < t_{E@39}$. Therefore, by (4.15) and (4.18), we have

$$H_p[q] = (i^*, s^*) \text{ throughout } [t_{W@18}, t_{E@39}]. \tag{4.22}$$

Since $t_{W@18} < t_{W@rsp}$ and $t_{W'@inv} < t_{W'@14} < t_{E@35}$, we have

$$H_p[q] = (i^*, s^*) \text{ throughout } [t_{W@rsp}, t_{W'@inv}]. \tag{4.23}$$

By (4.15) and (4.16), p's local variable f has the same value i^* during both W' and W, thus, Claim 4.21 applies. By Claim 4.21(a), p completes at least $2n+2$ `recycle()` calls

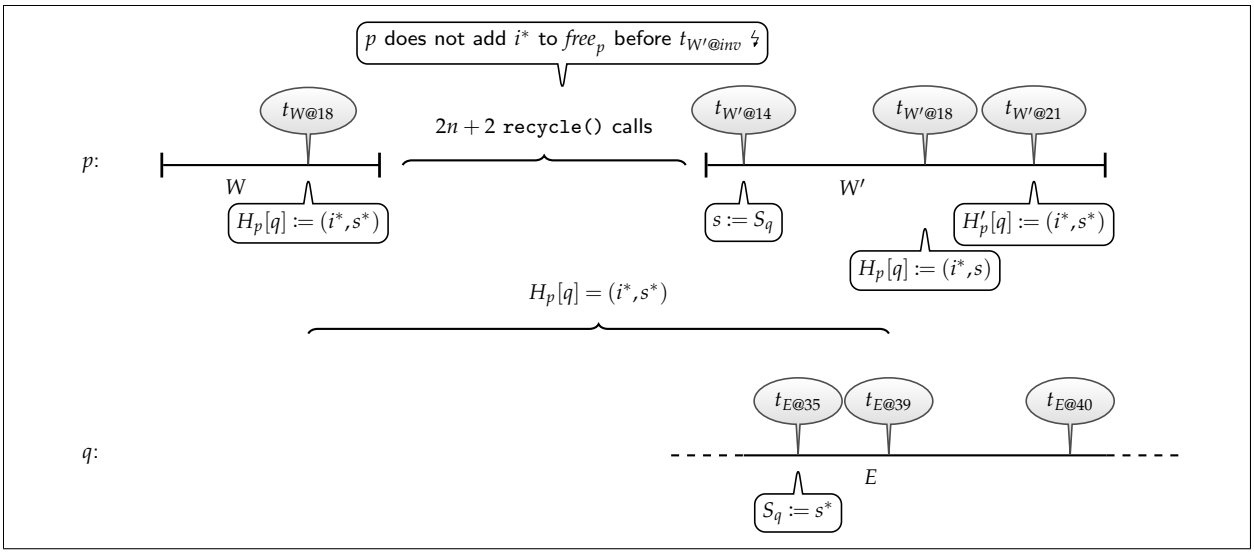

Figure 4.11: Illustration for the proof of (4.19) of Claim 4.25, Case 2.2 and if $t_{W'@14} < t_{E@35}$

$R_1, \ldots, R_{2n+2}$ during $[t_{W@rsp}, t_{W'@inv}]$. Therefore by Claim 4.17 and (4.23), p does not add i^* to $free_p$ throughout $[t_{R_n@rsp}, t_{W'@inv}]$. This is a contradiction, because by Claim 4.21(b), process p adds i^* to $free_p$ at some point during $[t_{R_{2n+2}@inv}, t_{W'@inv}) \subseteq [t_{R_n@rsp}, t_{W'@inv}]$. □

By Claim 4.24, if a process q obtains an index i^* indirectly during an `Exec()` call E, then, there is a `Write()` call W by some process p in which p writes i^* to both hint entries of q, at some point during E, before q reads the second hint. Therefore p also writes this index to Ptr during E. We show in the following claim, that W linearizes after q updates S_q during E. The idea is that if p writes to Ptr before that, then S_q must have wrapped around since p provides its first hint during W, however, then q must have reset the value stored in the first hint before q reads it, which is a contradiction. Moreover, we prove that the value of the first hint does not change before p writes to the second hint during W.

Claim 4.25. *Consider an* `Exec()` *operation call E by some process q during which q write some value s^* to S_q in line 35. Suppose process q obtains some index $i^* \in I_p$ indirectly, and executes $A[i^*]$.*`Oper(`*args*`)` *during E. Then there exists a process p and a* `Write()` *operation call W by p, such that W has the following properties:*

(a) process p writes i^* into Ptr at $t_{W@20}$,

(b) $t_{W@20}, t_{W@21} \in [t_{E@35}, t_{E@40}]$, and

(c) $H_p[q] = H'_p[q] = (i^*, s^*)$ at $t_{W@21}$.

Proof. Since q obtains i^* indirectly, q reads (i^*, s^*) from both $H_p[q]$ and $H'_p[q]$ in line 39 respectively line 40 of E. Thus by Claim 4.24, there is a `Write()` operation call W by p such that

$$p \text{ writes } (i^*, s^*) \text{ to } H_p[q], \text{ respectively } H'_p[q], \text{ at } t_{W@18}, \text{ respectively } t_{W@21}, \text{ and} \tag{4.24}$$

$$t_{W@21} < t_{E@40}, \text{ and} \tag{4.25}$$

$$p \text{ does not write } (i^*, s^*) \text{ to } H_p[q] \text{ during } (t_{W@18}, t_{E@40}]. \tag{4.26}$$

Since q reads (i^*, s^*) from $H_p[q]$ at $t_{E@39}$, by (4.26),

$$\text{if } t_{W@18} < t_{E@39}, \text{ then no process writes to } H_p[q] \text{ throughout } (t_{W@18} < t_{E@39}]. \tag{4.27}$$

Proof of Part (a). By (4.24) and by the implementation, process p writes i^* to Ptr in line 20 of W at point $t_{W@20}$.

Proof of Part (b). By (4.25), we have $t_{W@20} < t_{W@21} < t_{E@40}$. Hence, it suffices to show that $t_{W@20} \geq t_{E@35}$. By (4.24), process p writes the pair (i^*, s^*) to $H'_p[q]$ at $t_{W@21}$, hence we have,

$$S_q = s^* \text{ at } t_{W@19}. \tag{4.28}$$

Now suppose for the sake of contradiction that $t_{W@20} < t_{E@35}$ as depicted in Figure 4.12. This implies that $t_{W@20} < t_{E@39}$, so (4.27) holds. In line 34 of `Exec()`, q reads and increments modulo $2n$ the value stored in S_q, and in line 35, q writes the result to S_q, and S_q is not modified

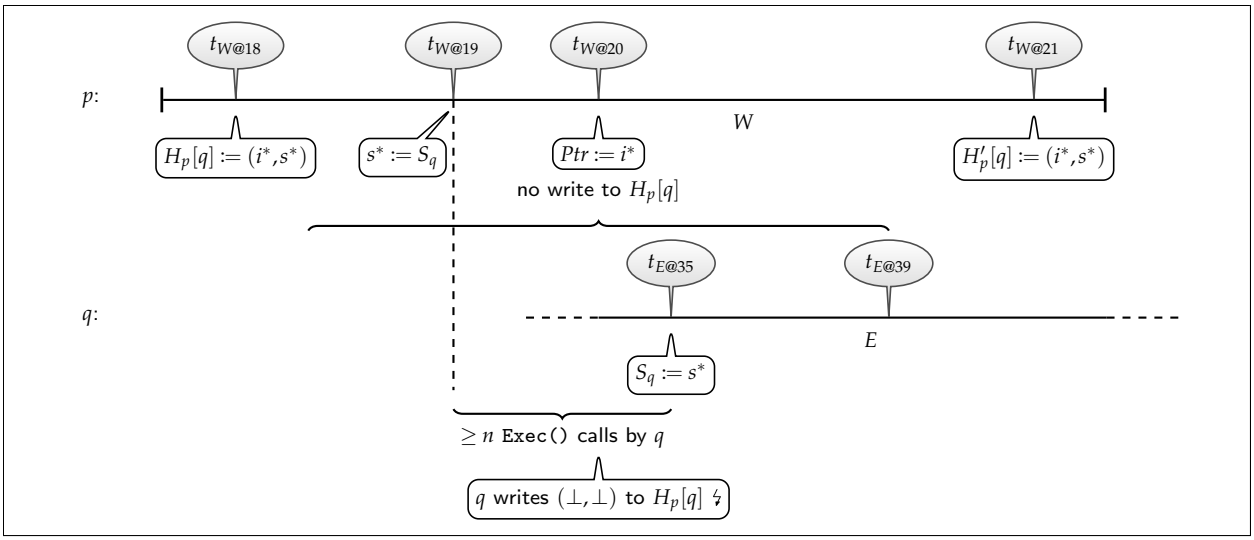

Figure 4.12: Illustration for the proof of Claim 4.25, Part (b)

in any other operation. Thus just before $t_{E@35}$, $S_q \neq s^*$. By the assumption $t_{W@20} < t_{E@35}$ and so $t_{W@19} < t_{E@35}$, and thus by (4.28), q must have executed line 35 of `Exec()` operation $2n$ times during time interval $[t_{W@19}, t_{E@35}]$. This implies that line 44 gets executed at least n times by q during $(t_{W@19}, t_{E@35})$. Thus, at least once when q executes line 44, $S_q \bmod n = p$, and so at least once process q writes the pair $(\perp, \perp)$ into $H_p[q]$ during $(t_{W@19}, t_{E@35})$, and therefore during $(t_{W@18}, t_{E@39})$. This contradicts (4.27).

Proof of Part (c). By (4.24), $H'_p[q] = (i^*, s^*)$ at point $t_{W@21}$. Now we show that $H_p[q] = (i^*, s^*)$ at the same point. Suppose for the purpose of contradiction that $H_p[q] \neq (i^*, s^*)$ at point $t_{W@21}$. By (4.24), $H_p[q] = (i^*, s^*)$ at point $t_{W@18}$. This implies that some process (p or q) writes to $H_p[q]$ during $(t_{W@18}, t_{W@21}]$. Process p does not write to $H_p[q]$ in lines 19–21. Therefore, process q writes $(\perp, \perp)$ to $H_p[q]$ in line 44 of some `Exec()` call E' at point $t_{E'@44} \in (t_{W@18}, t_{W@21}]$. Both, E and E' are executed by q, and $t_{E'@44} \leq t_{W@21}$, and thus by (4.25), we have $t_{E'@44} \leq t_{E@40}$. Hence, $E \neq E'$ and E' must be executed before E. Thus, we have $t_{W@18} < t_{E'@44} < t_{E@inv} < t_{E@39}$. This contradicts (4.27), because $t_{W@18} < t_{E@39}$, but q writes $(\perp, \perp)$ to $H_p[q]$ at $t_{E'@44} \in (t_{W@18}, t_{E@39})$. □

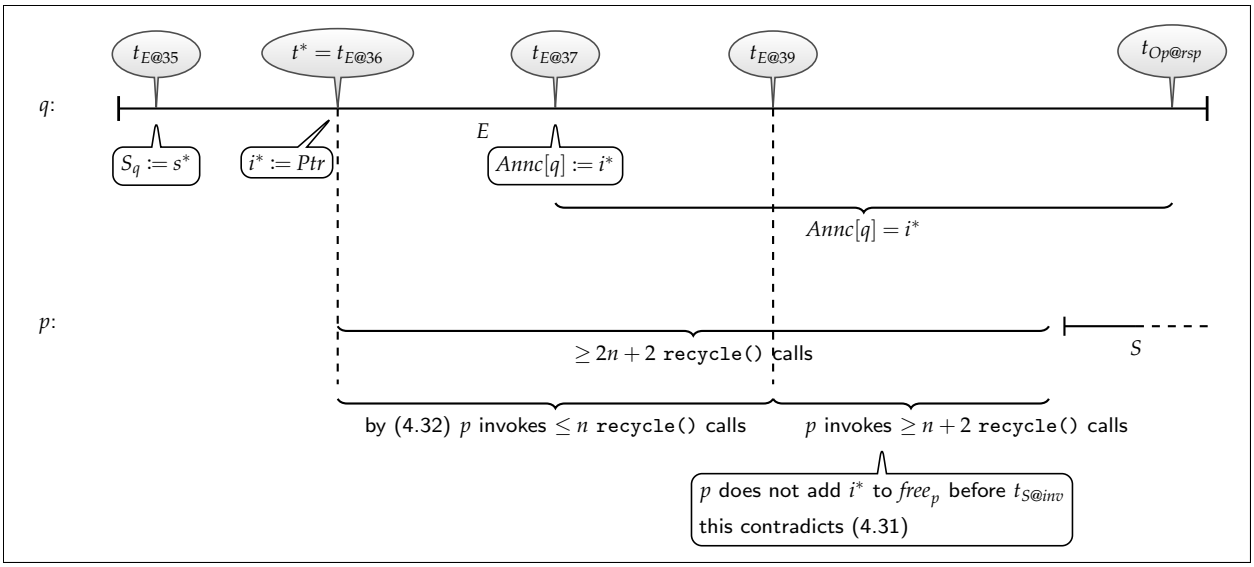

Figure 4.13: Illustration for the proof of Lemma 4.8, Case 1, Part (b)

Proof of Lemma 4.8 Utilizing all the results we obtain from these claims, we can finally prove Lemma 4.8 (as repeated in the following).

Lemma 4.8. *[Restated] Suppose some process q executes an* `Exec()` *operation call E in Λ, in which q invokes an $A[i^*]$.*`Oper()` *operation Op, for some i^*. Then there exists a point $t^* \in [t_{E@inv}, t_{Op@inv}]$, such that:*

(a) $Ptr = i^*$ at t^*, and

(b) no $A[i^*]$.`SWrite()` *operation call overlaps with interval $[t^*, t_{Op@rsp}]$.*

Proof. Let p be the owner of index i^*. There are two cases depending on whether q obtains i^* directly or indirectly.

Case 1: Process q obtains i^* directly. This implies that q reads i^* from Ptr at $t_{E@36}$, and then it writes i^* to $Annc[q]$ at point $t_{E@37}$. Since this is the only write to $Annc[q]$ during E,

$$Annc[q] = i^*, \text{ throughout } [t_{E@37}, t_{Op@rsp}]. \tag{4.29}$$

Let $t^* = t_{E@36}$, thus, $t^* \in [t_{E@inv}, t_{Op@inv}]$. Hence Part (a) is true, and so it remains to show Part (b). Assume for the sake of contradiction that there is an $A[i^*]$.`SWrite()` operation call S

which overlaps with time interval $[t^*, t_{Op@rsp}] = [t_{E@36}, t_{Op@rsp}]$ as shown in Figure 4.13. Only the owner of index i^*, process p, can execute an $A[i^*]$.`SWrite()` call, and so S is executed by p. By Claim 4.23, no $A[i^*]$.`SWrite()` operation call can be pending at $t_{E@36}$, thus, S gets invoked at $t_{S@inv} \in (t_{E@36}, t_{Op@rsp}]$. Since $Ptr = i^*$ at $t_{E@36}$, by Claim 4.22

$$\text{process } p \text{ completes at least } 2n+2 \texttt{ recycle()} \text{ calls during } [t_{E@36}, t_{S@inv}], \text{ and} \tag{4.30}$$

let R_{2n+1} be the $2n+1$-st `recycle()` call by p after $t_{E@36}$, then

$$p \text{ executes } free_p.\texttt{add}(i^*) \text{ at some point during } [t_{R_{2n+1}@inv}, t_{S@inv}) \tag{4.31}$$

In the following, we first prove that

$$p \text{ invokes at most } n \texttt{ recycle()} \text{ calls during } [t_{E@36}, t_{E@39}]. \tag{4.32}$$

This together with (4.30) imply that $t_{E@39} < t_{S@inv}$, and also that p invokes at least $n+2$ `recycle()` calls during $(t_{E@39}, t_{S@inv})$. Then we show that this contradicts (4.31).

Suppose (4.32) is not true, i.e. p invokes more than n `recycle()` operation calls during $[t_{E@36}, t_{E@39}]$. Process p invokes a `Write()` call with every invocation of a `recycle()` call, thus this implies that p invokes more than n `Write()` calls during this interval, and so q completes at least n `Write()` calls during this interval. In line 24 of each `Write()` operation call, p increments ctr_p modulo n, therefore, p completes at least one `Write()` operation call during $[t_{E@36}, t_{E@39}]$, such that $ctr_p = q$ at the invocation this `Write()` call. Let W' be the last such `Write()` operation call during $[t_{E@36}, t_{E@39}]$. By Claim 4.20, $H_p[q] = H'_p[q] = (j, s^*)$ at the response of W', for some value j, where s^* is the value q writes to S_q at $t_{E@35}$.

Since, $H_p[q] = H'_p[q] = (j, s^*)$ and $S_q = s^*$ at $t_{W'@rsp} \in [t_{E@36}, t_{E@39}]$, by Claim 4.18 (using $t = t_{W'@rsp}$) process p writes to neither $H_p[q]$ nor $H'_p[q]$ between $t_{W'@rsp}$ and the first point when the value stored in S_q changes or q writes to $H_p[q]$ or $H'_p[q]$. Process q does not write to either $H_p[q]$ or $H'_p[q]$ during $[t_{E@36}, t_{E@40}]$, and so it does not write to either $H_p[q]$ or $H'_p[q]$ during $[t_{W'@rsp}, t_{E@40}]$. Moreover, the value stored in S_q does not change during the same interval.

Hence Claim 4.18 implies that p does not write to $H_p[q]$ and $H'_p[q]$ throughout $[t_{W'@rsp}, t_{E@40}]$, thus

$$H_p[q] = H'_p[q] = (j, s^*) \text{ throughout } [t_{W'@rsp}, t_{E@40}]. \tag{4.33}$$

Because $t_{W'@rsp} \leq t_{E@39}$, by (4.33) the pair q reads from both $H_p[q]$ and $H'_p[q]$ at $t_{E@39}$ and $t_{E@40}$, respectively, is (j, s^*). Thus, the if-condition in line 41 of E evaluates to true, and so q executes line 42. This is a contradiction, because we assumed q obtains i^* directly. This proves (4.32).

Thus by (4.32) and (4.30), p invokes at least $n+2$ `recycle()` calls $R_1, \ldots, R_{n+2}$ during $(t_{E@39}, t_{S@inv}]$. In the following, we show how this contradicts (4.31), and therefore, Part (b) is proved to be correct for the case that q obtains i^* directly.

By (4.29) $Annc[q] = i^*$ throughout $(t_{E@39}, t_{S@inv}] \subseteq [t_{E@37}, t_{Op@rsp}]$. Moreover, `recycle()` calls $R_1, \ldots, R_n$ all complete during $(t_{E@39}, t_{S@inv}]$. Thus by Claim 4.17, p does not add i^* to $free_p$ throughout $[t_{R_n@rsp}, t_{S@inv}]$. By (4.32), p invokes at most n `recycle()` calls before R_1, and so $t_{R_n@rsp} < t_{R_{2n+1}@inv}$, where R_{2n+1} is the $2n+1$-st `recycle()` call by p after $t_{E@36}$. Therefore, p does not add i^* to $free_p$ during $[t_{R_{2n+1}@inv}, t_{S@inv}]$, which contradicts (4.31).

Case 2: Process q obtains i^* indirectly. Therefore, Claim 4.25 holds. By Claim 4.25 (a), there exists a `Write()` operation call W by the owner of index i^*, process p, during which p writes i^* into Ptr at point $t_{W@20}$. We prove Parts (a) and (b) for $t^* = t_{W@20}$. By Claim 4.25 (b), $t_{W@20} \in [t_{E@35}, t_{E@40}] \subseteq [t_{E@inv}, t_{Op@inv}]$, thus Part (a) is correct.

Since $i^* \in I_p$, only process p can execute $A[i^*]$.`SWrite()`. To prove Part (b), we assume for the sake of contradiction that there is an $A[i^*]$.`SWrite()` operation call S by p which overlaps with time interval $[t_{W@20}, t_{Op@rsp}]$ as shown in Figure 4.14. By Claim 4.23, no $A[i^*]$.`SWrite()` operation call can be pending at point $t_{W@20}$, and p does not invoke any `SWrite()` during $[t_{W@20}, t_{W@rsp}]$. Thus, S is invoked at some point after $t_{W@rsp}$ and before $t_{Op@rsp}$, i.e.

$$t_{S@inv} \in (t_{W@rsp}, t_{Op@rsp}]. \tag{4.34}$$

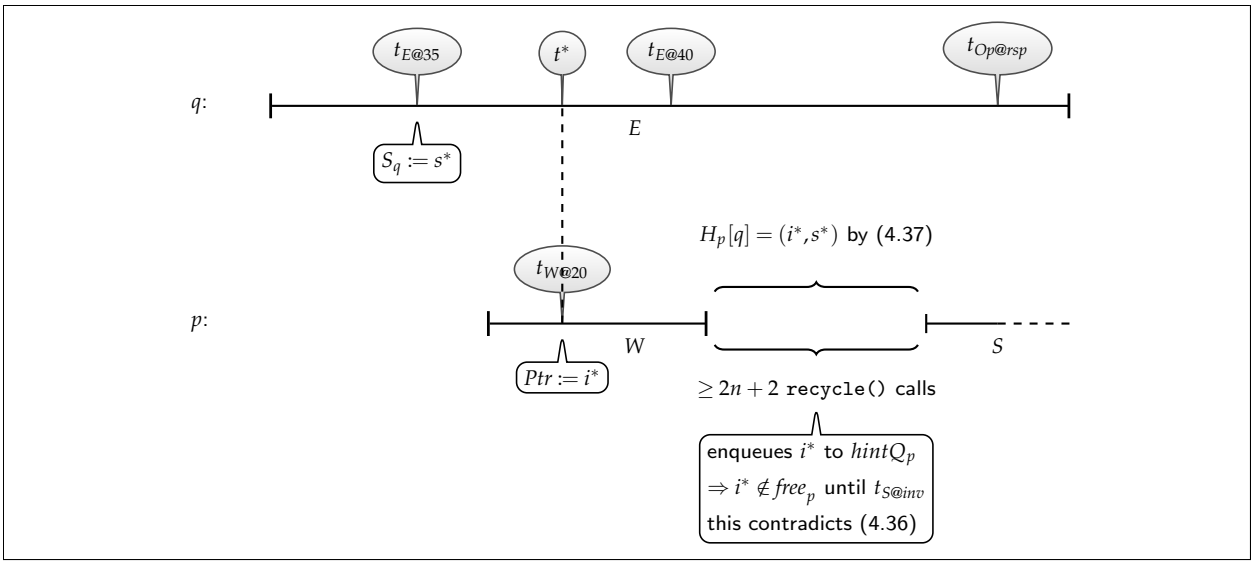

Figure 4.14: Illustration for the proof of Lemma 4.8, Case 2, Part (b)

Since $Ptr = i^*$ at $t_{W@20}$, by Claim 4.22 and because p does not execute any `recycle()` calls between line 20 and the response of W, we can conclude that

$$\text{process } p \text{ completes at least } 2n+2 \texttt{ recycle()} \text{ calls } R_1, \ldots, R_{2n+2} \text{ during interval } [t_{W@rsp}, t_{S@inv}) \subseteq [t_{W@20}, t_{S@inv}), \text{ and} \tag{4.35}$$

$$p \text{ executes } free_p.\texttt{add}(i^*) \text{ at some point during } [t_{R_{2n+1}@inv}, t_{S@inv}) \tag{4.36}$$

In the following, we first show that

$$H_p[q] = H'_p[q] = (i^*, s^*) \text{ throughout } [t_{W@21}, t_{S@inv}], \tag{4.37}$$

where s^* is the value q writes to S_q at point $t_{E@35}$. Next we show how this contradicts (4.36), and so the correctness of Part (b) is established.

Now we prove that (4.37) is true. By Claim 4.25 (c), we have $H_p[q] = H'_p[q] = (i^*, s^*)$ at $t_{W@21}$. Moreover, q writes s^* to S_q at $t_{E@35}$, and it does not change this value before E responds. Hence, $S_q = s^*$ throughout $[t_{E@35}, t_{E@40}]$, and so $S_q = s^*$ at $t_{W@21}$, because by Claim 4.25 (b), $t_{W@21} \in [t_{E@35}, t_{E@40}]$. Therefore, by Claim 4.18, process p does not write to

either $H_p[q]$ or $H'_p[q]$, in the interval between $t_{W@21}$ and the point at which q writes to any of $H_p[q]$, $H'_p[q]$, and S_q. Process q writes to neither $H_p[q]$ nor $H'_p[q]$ during $[t_{E@35}, t_{Op@rsp}]$, and $S_q = s^*$ throughout the same interval. Therefore q does not write to any of these three registers during $(t_{W@21}, t_{Op@rsp}] \subseteq [t_{E@35}, t_{Op@rsp}]$. So Claim 4.18 implies that p also does not write to any of $H_p[q]$ and $H'_p[q]$ during $(t_{W@21}, t_{Op@rsp}]$. Hence, $H_p[q] = H'_p[q] = (i^*, s^*)$ throughout $[t_{W@21}, t_{Op@rsp}]$. By (4.34), we have $[t_{W@21}, t_{S@inv}] \subseteq [t_{W@21}, t_{Op@rsp}]$, and so (4.37) follows.

Now consider the first n `recycle()` calls $R_1, \ldots, R_n$ of (4.35), executed by p during $[t_{W@rsp}, t_{S@inv})$. By (4.37), $H_p[q] = (i^*, s^*)$ throughout the same interval. Therefore by Claim 4.17, p does not add i^* to $free_p$ during $[t_{R_n@rsp}, t_{S@inv}]$. So p does not add i^* to $free_p$ during $[t_{R_{2n+1}@inv}, t_{S@inv}]$, which contradicts (4.36). □

4.4.3 Proof of Theorem 4.1

Our implementation of Section 4.3 follows the template of Figure 4.4, and so by Lemmas 4.5 and 4.8, it provides a linearizable implementation of any writable type $\mathcal{T}$ from instances of sequentially writable object of the same type and registers. Each `Write()` and `Exec(Oper,`arg`)` operation call on the resulting object executes a constant number of steps in addition to one `SWrite()` and `Oper()` call, respectively, executed on an instance of a sequentially writable object of the same type. Given the size of the data structures depicted in Figure 4.5 and the result of Claims 4.11 and 4.14, it follows that our implementation requires $2n^2 + 2n + 1$ shared registers and $n(8n + 9)$ instances of $\mathcal{SW}$ object. Each register needs to store at most $O(n^2)$ different values, and therefore the registers used for this implementation are of size $2\log n + O(1)$ bits. This completes the proof of Theorem 4.1.

4.5 Optimal-Time Sequentially Resettable (k,b)-Array

Recall that the resettable type (k,b)-Array stores an array S of size k, and each array entry $S[i]$ stores a b-bit value, for $i \in \{0, \ldots, k-1\}$. This type supports three operations: `Read()`,

`Write()`, and `Reset()`. For each $i \in \{0,\ldots,k-1\}$, a `Read(`i`)` returns the value of the i-th entry of the array, and a `Write(`i,x`)` writes value x to $S[i]$. A `Reset()` operation resets $S[i]$ to its initial value, for all $i \in \{0,\ldots,k-1\}$.

A straightforward way to implement a sequentially resettable (k,b)-Array from k registers is to write the initial value of the register into each array element during the `SReset()` operation. (Recall that an `SReset()` is a `Reset()` operation that has to be executed in isolation in order to guarantee the linearizability of the object.) However, the step complexity of such `SReset()` operation is proportional to the size of the array. This section provides a sequentially resettable implementation of this object from $k+1$ registers, such that each operation requires a constant number of steps. In this section, we assume $\perp$ denotes the initial value of a register.

4.5.1 The Implementation

In this implementation, processes share a register *Ver*, which stores a version number, and an array $A[0\ldots k-1]$ of registers, where each array entry $A[i]$ stores a pair. First we explain the implementation assuming *Ver* is an unbounded register. To execute an S.`SReset()`, i.e. to sequentially reset the entire array S, a process simply reads and increments the version number stored in *Ver*. Because `SReset()` is constrained to not overlap any other operation, these two steps of this operation do not overlap any other operation calls, and so `SReset()` correctly increments the value stored in *Ver*. To execute a `Write(`i,x`)` operation, the calling process reads *ver* from *Ver* and then writes pair (x,ver) into $A[i]$. To return the value of the i-th element of S in a `Read(`i`)`, the calling process reads the pair (x,v) stored in $A[i]$, and then it checks whether v is the same version number as the one stored in *Ver*. If yes, then it returns x, and otherwise, it returns the initial value of the register.

In order to bound register *Ver*, we increment the version number modulo k during each S.`SReset()` operation call. However, this can cause the following problem. Suppose a process reads some value $(\cdot,v)$ from a register $A[i]$, $i \in \{0,\ldots,k-1\}$ and then checks that v is the value of *Ver*. This process cannot distinguish between the case that a process wrote to A since the last

```
shared
  int Ver = 0
  Array<register> A[0,...,k-1] = ((⊥,⊥),···,(⊥,⊥))

Operation SReset()
56 v := (Ver.Read() + 1) mod k
57 Ver.Write(v)
58 A[v].Write(⊥,⊥)

Operation Read(i)
59 (x,v) := A[i].Read()
60 ver := Ver.Read()
61 if v ≠ ver then x := ⊥
62 return x

Operation Write(i,x)
63 ver := Ver.Read()
64 A[i].Write(x,ver)
```

Figure 4.15: Implementation of a Sequentially Resettable (k,b)-Array

time $Ver = v$, or the value stored in Ver has wrapped around at least once since the last write to $A[i]$. To avoid this problem, we ensure that during any sequence of k consecutive S.`SReset()` executions, each register in A is actually reset once. We achieve this by resetting one register of A in each `SReset()` operation call, thus ensuring that all registers of A are reset after every k consecutive S.`SReset()` operation calls. This implementation is depicted in Figure 4.15.

4.5.2 Proof of Lemma 4.2

This section provides a proof for Lemma 4.2 (as repeated in the following).

Lemma 4.2 (Restated)**.** *There is an implementation of a sequentially resettable (k,b)-Array, from $k+1$ registers, each of size $b + O(\log k)$ bits, in which each* `Read()`, `Write()` *and* `SReset()` *operation call has constant step complexity.*

Consider the implementation of Figure 4.15. Each operation consists of a constant number of shared steps. Moreover, this implementation uses $k+1$ registers, and each register stores at most $O(\log k)$ bits in addition to the data value it stores. Hence to prove this lemma, it is enough

to show that any transcript Λ obtained by executing operations implemented as in Figure 4.15 is linearizable, provided no `SReset()` call overlaps any other operation call in Λ.

We define a linearization point $lin(Op)$ for each operation call Op in $\Gamma(\Lambda)$ by their linearization points. If Op is a `Read()` call, then we let $lin(Op) = t_{Op@inv} = t_{Op@59}$. If Op is a `Write()` call, then we let $lin(Op) = t_{Op@rsp} = t_{Op@64}$. Finally, for an `SReset()` call Op, we let $lin(Op) = t_{Op@rsp}$. Clearly, for any operation call Op, we have $lin(Op) \in [t_{Op@inv}, t_{Op@rsp}]$.

Let S_Λ be the sequential history obtained from ordering operation calls in $\Gamma(\Lambda)$. To prove that S_Λ is valid, it is enough to show that for any `Read(`i^*`)` operation call R,

(a) if no `Write(`i^*`,·)` call proceeds R in S_Λ, then R returns $\perp$,

(b) otherwise, let $W =$ `Write(`i^*`,`x_W`)` be the last `Write(`i^*`,·)` operation call that precedes R in S_Λ, then

  (b1) if there is no `SReset()` call between W and R in S_Λ, then R returns x_W, and

  (b2) R returns $\perp$, otherwise.

Proof of Part (a). If no `Write(`i^*`,·)` call appears before R in S_Λ, then no process executes line 64 for $i = i^*$. Therefore, Part (a) immediately follows, because initially $A[i^*] = (\perp, \perp)$ and has not changed, and $Ver \geq 0$, and so the if-condition in line 61 of R evaluates to true.

Proof of Part (b). Now suppose $W =$ `Write(`i^*`,`x_W`)` is the last `Write(`i^*`,·)` call before R in S_Λ. Then Parts (b1) and (b2) follow from the following claim.

Let p be the process that executes W, and let (x_W, v_W) be the pair that p writes to $A[i^*]$ in line 64 of W, for some v_W. Also let q be the process that executes R. No S.`SReset()` call overlaps any other operation calls in Λ, therefore

$$\text{no } S.\texttt{SReset()} \text{ operation call overlaps } W \text{ or } R. \tag{4.38}$$

Proof of Part (b1). First suppose no S.`SReset()` operation call appears between W and R in S_Λ. Therefore by (4.38), no S.`SReset()` operation call overlaps interval $[t_{W@inv}, t_{R@rsp}]$. Operation call W is the last S.`Write(`i^*`,·)` call that precedes R in S_Λ, so no process executes line 64 during $[t_{W@rsp}, t_{R@inv}] = [t_{W@64}, t_{R@60}]$. Thus the value of $A[i^*]$ does not change throughout this interval. Hence

$$q \text{ reads } (x_W, v_W) \neq (\perp, \perp) \text{ from } A[i^*] \text{ in line 60 of } R. \tag{4.39}$$

Moreover, as the value stored in *Ver* only changes during an S.`SReset()` operation call, and no S.`SReset()` operation call overlaps interval $[t_{W@inv}, t_{R@rsp}]$, *Ver* stores the same value throughout this interval. Recall that p writes (x_W, v_W) to $A[i^*]$ in W, thus, p reads v_W from *Ver* in line 63 of W. Thus, $Ver = v_W$ throughout $[t_{W@inv}, t_{R@rsp}]$, and in particular when p reads this variable in line 60 of R. Thus, when q evaluates the if-condition in line 61 of R, its local variable *ver* has value v_W, and so because of (4.39) the if-condition evaluates to false. Therefore, operation call R returns $x = x_W$.

Proof of Part (b2). Next suppose that an S.`SReset()` operation call appears between W and R in S_Λ. Let Rs be the last such S.`SReset()` call, hence by (4.38), Rs is the last S.`SReset()` call that completes during $[t_{W@rsp}, t_{R@inv}]$. Since the value of *Ver* can only change during an `SReset()` operation call,

$$\text{process } q \text{ reads } v^* \text{ from } Ver \text{ in line 60 of } R, \tag{4.40}$$

where $v^* \neq \perp$ is the value written to *Ver* in line 57 of Rs. By the assumption, W is the last S.`Write(`i^*`,·)` call that precedes R in S_Λ, and (x_W, v_W) is the pair written to $A[i^*]$ during this operation call. Therefore when q reads some pair (x_R, v_R) from $A[i^*]$ in line 59 of R, either $v_R = \perp$ if a process writes $(\perp, \perp)$ to $A[i^*]$ in an `SReset()` call during interval $[t_{W@rsp}, t_{R@inv}]$, or $v_R = v_W$ otherwise. If $v_R = \perp$, then by (4.40), $v_R \neq v^*$, and so the if-condition of line 61 evaluates to true. Thus, R returns $\perp$.

Now suppose $v_R = v_W$, hence, no process writes $(\perp, \perp)$ to $A[i^*]$ throughout $[t_{W@rsp}, t_{R@inv}]$. If $v_R = v_W \neq v^*$, then the if-condition is satisfied, and R returns $\perp$. Finally, we show that it is not possible that $v_R = v_W = v^*$, by assuming for the sake of contradiction that $v_R = v_W = v^*$. Operation call Rs completes during $[t_{W@rsp}, t_{R@inv}]$. The value of *Ver* is incremented to v^* in Rs at some point during interval $[t_{W@rsp}, t_{R@inv}]$. Moreover, since p writes (x_W, v_W) to $A[i^*]$ in W, it must have read $v_W = v^*$ in line 63 of W, and by (4.40), process q also reads v^* from *Ver* in line 60 of R. Thus, the value stored in *Ver* wraps around during $[t_{W@rsp}, t_{R@inv}]$. This implies that k `SReset()` operation calls complete during this interval, and therefore, line 58 gets executed for all $v \in \{0, \ldots, k-1\}$. Therefore, some process writes $(\perp, \perp)$ to $A[i^*]$ at some point during $[t_{W@rsp}, t_{R@inv}]$, which is a contradiction.

This concludes the proof of Part (b), therefore S_Λ is a linearization of Λ.

Chapter 5

ABA-Detecting Registers

5.1 Results

To investigate the complexity of detecting or preventing ABAs, we define a natural object, the *ABA-detecting register* . It supports two operations, `DRead()` and `DWrite()`. Operation `DWrite(`x`)` writes value x to the register, and returns nothing. Operation `DRead()` by process p returns, in addition to the value of the register, a Boolean flag, which is true if and only if some process executed a `DWrite()` since p's last `DRead()` operation call or since the beginning of the execution, whichever is later. We distinguish between *single-writer* ABA-detecting registers, where only one dedicated process is allowed to call `DWrite()`, and *multi-writer* ones that don't have this restriction.

A wait-free ABA-detecting register can be implemented from registers. Therefore, they are weaker with respect to wait-freedom than CAS or LL/SC. Using a single unbounded register with an unbounded tag that gets changed whenever some process writes to it, it is trivial to obtain an ABA-detecting register with constant step complexity. But if we restrict ourselves to bounded base objects, the situation is completely different.

In Section 5.2, we propose an implementation of an ABA-detecting register that uses $n+1$ bounded registers, and each operation requires a constant number of steps. More specifically,

Theorem 5.1. *There is a linearizable wait-free implementation of a multi-writer b-bit ABA-detecting register from $n+1$ registers, each of size $(b+2logn+O(1))$ bits with constant step complexity.*

The number of registers used in this implementation is optimal (up to a constant additive

term) based on the lower bound by Aghazadeh and Woelfel (2015)[1]. They show that any implementation of a single-writer ABA-detecting register in a system with n processes and bounded registers, requires at least $n-1$ registers, even if the implementation satisfies only nondeterministic solo-termination (the non-deterministic variant of obstruction-freedom), which is a progress condition strictly weaker than wait-freedom.

Based on their lower bounds, the availability of CAS is of little help in terms of time-space tradeoff: For wait-free implementations from CAS objects and registers, there is a time-space tradeoff that is linear in n (Aghazadeh and Woelfel, 2015). The same asymptotic time-space tradeoff can be obtained, if the base objects support arbitrary conditional read-modify-write operations as defined by Fich, Hendler and Shavit (2006). Each conditional operation can be simulated by a single operation on a *writable* CAS object. For that reason, Aghazadeh and Woelfel (2015) state their lower bound for implementations from conditional read-modify-write operations in terms of writable CAS base objects. A summary of their lower bounds is provided here:

Theorem 5.2. *(Aghazadeh and Woelfel, 2015) Any linearizable implementation of a single-writer 1-bit ABA-detecting register from m base objects satisfies:*

(a) $m \geq n-1$, if the base objects are bounded registers, and the implementation satisfies nondeterministic solo-termination,

(b) $m \geq (n-1)/t$, if the the base objects are bounded CAS objects and registers, and the implementation is deterministic and wait-free with worst-case step complexity at most t, and

(c) $m \geq (n-1)/(2t)$, if the base objects are bounded writable CAS objects, and the implementation is deterministic and wait-free with worst-case step complexity at most t.

The requirement that base objects are bounded is necessary for these lower bounds, because,

[1] The lower bound results in this publication should not be considered contributions to this thesis. Therefore proofs of the lower bounds are omitted here.

as mentioned earlier, an ABA-detecting register can be trivially obtained by augmenting a normal register with an unbounded tag.

In Section 5.3, we build an ABA-detecting register from a single LL/SC/VL object of the same size, and we prove the following.

Theorem 5.3. *There is an implementation of an ABA-detecting register from a single LL/SC/VL object of the same size, such that each* `DRead()` *and* `DWrite()` *operation requires at most two shared memory steps.*

Thus, by reduction we obtain the same lower bound as the one stated in Theorem 5.2 for implementations of single-bit LL/SC/VL objects. Unfortunately, for that reduction the `VL()` operation is needed, and at least we do not know how to obtain a similarly efficient ABA-detecting register from an LL/SC object that does not support `VL()`. However, Aghazadeh and Woelfel (2015) discuss that their lower bound proofs of Theorem 5.2 can be easily modified to accommodate LL/SC objects:

Corollary 5.4. *(Aghazadeh and Woelfel, 2015) Any linearizable implementation of a single bit LL/SC object from m bounded base objects satisfies*

(a) $m \geq (n-1)/t$*, if the the base objects are bounded CAS objects and registers, and the implementation is deterministic and wait-free with worst-case step complexity at most t, and*

(b) $m \geq (n-1)/(2t)$*, if the base objects are bounded writable CAS objects, and the implementation is deterministic and wait-free with worst-case step complexity at most t.*

A linear space lower bound (corresponding to Part (a) of Theorem 5.2) for nondeterministic solo-terminating implementations of LL/SC from (even unbounded) registers follows from the fact that LL/SC objects are *perturbable* (Jayanti, Tan and Toueg, 2000).

As in Theorem 5.2, the assumption that base objects are bounded is necessary, because there is an implementation of an LL/SC/VL object from a single unbounded CAS object with constant

step complexity (Moir, 1997). The time-space tradeoff of Corollary 5.4(a) is asymptotically tight for implementations with constant step complexity, as it matches known upper bounds by Jayanti and Petrovic (2003). In Section 5.4, we show that it is also asymptotically tight for implementations using a single CAS object:

Theorem 5.5. *A single bounded CAS object of size $b + n$ bits suffices to implement a bounded LL/SC/VL object of size b bits with $O(n)$ step complexity.*

This in addition to Theorem 5.3 gives us the following result:

Corollary 5.6. *A single bounded CAS object suffices to implement a bounded multi-writer ABA-detecting register with $O(n)$ step complexity.*

Our upper bounds suggest that bounded CAS objects (and in fact any conditional read-modify-write operations) are not more helpful than bounded registers with respect to ABA detection. On the other hand, ABA detection is difficult only if base objects are bounded. The lower bounds of Aghazadeh and Woelfel (2015) clearly indicate that ABA detection is inherently difficult, even if bounded conditional read-modify-write primitives such as (writable) CAS objects are available. (For those lower bounds, it does not matter how large that bound on the size of the base object is, as long as it is finite.) Therefore, other common primitives that provide a solution to the ABA problem would most likely be as difficult to obtain as those that use LL/SC. Even though some of the results may be intuitive and not surprising, they give rigorous explanations of what makes ABA-detection difficult. Moreover, lower bounds of Aghazadeh and Woelfel (2015) also confirm that researchers will not be surprised in the future by finding better algorithms than the existing ones.

Our new ABA-detecting register has already been used in other algorithms (Aghazadeh and Woelfel, 2016). In Chapters 6 and 7, we demonstrate two different applications of this object.

```
shared:
register X = (⊥,⊥)
register Annc[0...n−1] = (⊥,...,⊥)

Operation DWrite_p(x)
 1 s := getSeq()
 2 X.Write(x,s)

Operation getSeq_p()
 3 a := Annc[ctr_p].Read()
 4 if a ∈ I_p then
 5     annc Q_p.enq(a)
 6 else
 7     annc Q_p.enq(⊥)
 8 annc Q_p.deq()
 9 ctr_p := (ctr_p + 1) mod n
10 choose arbitrary s ∈ I_p \ (annc Q_p ∪ used Q_p)
11 used Q_p.enq(s)
12 used Q_p.deq()
13 return s

local (to each process p):
Boolean b_p = false
Queue used Q_p, annc Q_p    // n+1 ⊥ values
are initially in each queue
int ctr_p = 0

Operation DRead_q()
14 (x,s) := X.Read()
15 a := Annc[q].Read()
16 Annc[q].Write(s)
17 (x',s') := X.Read()
18 if s = a then
19     ret := (x,b_q)
20 else
21     ret := (x,true)
22 if (x,s) = (x',s') then
23     b_q := false
24 else
25     b_q := true
26 return ret
```

Figure 5.1: An ABA-detecting Register Implemented from Bounded Registers.

5.2 Optimal Constant-Time ABA-Detecting Register from Registers

In this section, a linearizable implementation of an ABA-detecting register from $n+1$ bounded registers with constant step complexity is provided. This implementation uses two more registers than the lower bound by Aghazadeh and Woelfel (2015) stated in Theorem 5.2 (a).

The main idea of the algorithm is similar to the one used in the multi-layered construction of LL/SC/VL from CAS by Jayanti and Petrovic (2003), which itself is a modified version of the implementations by Anderson and Moir (1995) and Moir (1997). In the following, we discuss the implementation. A complete correctness proof is provided in Section 5.2.2.

5.2.1 Algorithm Description

Suppose each process p owns a set of sequence numbers I_p. In the naive approach $I_p = \{i \mid i \geq 0 \wedge i \bmod n = p\}$, and a shared unbounded register X is used to store a pair (x,s),

where x is the data value stored in the ABA-detecting register, and s is a sequence number. Each time a process p executes a `DWrite()` call, it increments the last sequence number it used by n and stores it alongside the data value in X. Hence, if a process reads (x,s) and (x,s') from X in two consecutive `DRead()` operations, for $s \neq s'$, then it knows that there was a `DWrite()` in between.

In our implementation, we avoid using unbounded sequence numbers, as explained in the following. We use a *bounded* register X, and we let $I_p = \{i \mid 0 \leq i < 2n(n+1) \wedge i \bmod n = p\}$. As in Section 4.3, each process, q, announces the last sequence number it reads from X by writing it into $Annc[q]$. Register X is initialized to $(\perp,\perp)$ and all entries of $Annc$ are initialized to $\perp$. The pseudocode for this algorithm is depicted in Figure 5.1.

During a `DWrite(`x`)` operation call, the calling process p writes (x,s) to X, where s is a sequence number in I_p. Moreover, for any $s^* \in I_p$ and any process q, our algorithm maintains the following invariant:

> ($\star$) If $X = (\cdot,s^*)$ at some point t, and $Annc[q] = s^*$ throughout $[t,t']$, for some $t' \geq t$, then p does not write $(\cdot,s^*)$ into X during $(t,t']$.

We say such sequence number s^* is *properly announced* by q throughout $[t,t']$.

To maintain ($\star$), a process writes some sequence number s into X only if it reads the entire announce array when X does not store sequence number s anymore, and it does not find s in any entry of this array. The `DWrite()` operation uses a helper function `getSeq()` to find such sequence number s. To achieve constant step complexity, similar to the `recycle()` operation of Section 4.3, in a sequence of n consecutive `getSeq()` calls process p scans through the entire announce array, reading one array entry with each `getSeq()` call. It then returns a sequence number that p has not used in its preceding n `DWrite()` operation calls, and which it has not found in any array entry of $Annc$, when it read that entry last. The `DWrite()` operation call linearizes at the point when p writes to X.

During `DRead()` call R, the calling process q reads some pair (x,s) from X, and writes s into $Annc[q]$. Process q has two tasks during this operation,

(I) ensures that s is properly announced, or remembers that it failed to do so, and

(II) decides a correct return value for its `DRead()` call.

If q ensures that s is properly announced during this `DRead()` call, then ($\star$) guarantees that the owner of s does not reuse this sequence number as long as s is stored in $Annc[q]$, so in particular, until q announces a sequence number during its next `DRead()` call. This is very helpful, when p has to decide the return value of R, as described later.

First we explain how q performs task (I). Imagine p could read s from X and announce it in one atomic step, then s would be immediately properly announced. But since this is not feasible, we employ the following idea. After q reads (x,s) from X and writes s to $Annc[q]$, process q reads X again into (x',s'). If q reads the same pair in both of those reads, i.e. $(x,s) = (x',s')$, then q knows that at the point of its second read, $X = (x,s)$ and $Annc[q] = s$. Thus, s is properly announced from that point until q announces another sequence number, so (I) is achieved in this case. Now suppose $(x,s) \neq (x',s')$. Then q cannot ensure that s was properly announced at some point during R. So q remembers this by setting a local boolean variable b_q to `true`. (This bit is always set to `false` at the invocation of each `DRead()` call by q.) So task (I) is done.

It is important to point out that when q reads different pairs from X, process q witnesses a write to X in between those reads of X. So while doing task (I), either q ensures that s is properly announced at the second read of X, or it witnesses a write to X, and sets b_q to `true`. In the former case, R linearizes at the second read of X, which is at the response of R, and in the latter case, R linearizes at the first read of X, which is at the invocation of R. Thus, our algorithm maintains the following invariant for each `DRead()` call R by q that linearizes at some point $lin(R)$.

$$b_q = \texttt{true} \text{ at } t_{R@rsp}, \text{ if and only if a } \texttt{DWrite()} \text{ linearizes during } (lin(R), t_{R@rsp}]. \tag{5.1}$$

To perform task (II), process q has to choose a pair (x^*,g^*) as the return value for its `DRead()` call, R. Process q chooses x^* to be the value that it reads from the first component of

X at its linearization point. That is $x^* = x$, if $lin(R) = t_{R@inv}$, and $x^* = x'$, if $lin(R) = t_{R@rsp}$. It is interesting to point out that in the second case, since q reads the same pair from X in its both reads of X, $x^* = x' = x$.

The only remaining job for q is to determine the value of g^*. Let R' be q's last `DRead()` call prior to R. If $b_q = \texttt{true}$, then by (5.1), a `DWrite()` has linearized during $(lin(R'), t_{R'@rsp}]$, and so during $(lin(R'), lin(R)]$. So q chooses $g^* = \texttt{true} = b_q$.

If $b_q = \texttt{false}$, then q has ensured that the sequence number, a, that q announced during R', was properly announced at $lin(R')$. Thus, $(\star)$ guarantees that the owner of sequence number a does not write a again into X between $lin(R')$ and the point at which q announces another sequence number. So if $s = a$, then q announces the same sequence number during R, and so $a = s$ remains properly announced until at least the response of R'. So reading the same sequence number as a during R implies that no write to X occurs during $[lin(R'), t_{R@rsp})$, and so no `DWrite()` linearizes during $[lin(R'), lin(R))$. Thus, q returns $g^* = \texttt{false} = b_q$. Now suppose $s \neq a$. As we discussed a is properly announced at $lin(R')$ in this case. So $X = (\cdot, a)$ at $lin(R')$, and since q reads $X = (\cdot, s)$ at $t_{R@inv}$, the value of X must have changed during $[lin(R'), t_{R@inv})$. Therefore, a `DWrite()` has linearizes during $[lin(R'), lin(R))$, so q chooses $g^* = \texttt{true}$.

5.2.2 Proof of Theorem 5.1

Consider the implementation of the ABA-detecting register in Figure 5.1. Each operation requires a constant number of steps. Register X stores a sequence number in addition to the data, and registers of $Annc$ store only a sequence number. The sequence number is from set $\{0, \ldots, 2n(n+1) - 1\}$. So if the data is at most b bits, then it is enough to have registers of size $b + 2\log n + O(1)$ bits. Hence, to complete the proof of Theorem 5.1, we show that the implementation of Figure 5.1 is linearizable.

Consider a transcript Λ obtained by processes executing `DRead()` and `DWrite()` operation calls on an ABA-detecting register as implemented in Figure 5.1. We define a point $lin(Op)$ for

each operation call Op in $H = \Gamma(\Lambda)$. If Op is a `DWrite()` operation call, then we let $lin(Op) = t_{Op@2}$. If Op is a `DRead()` operation call by some process q, then we define $lin(Op) = t_{Op@14}$ if $b_q = \texttt{true}$ at $t_{Op@rsp}$, and otherwise $lin(Op) = t_{Op@17}$. Clearly, for any operation call Op, we have $lin(Op) \in [t_{Op@inv}, t_{Op@rsp}]$. To show that $lin(Op)$ is a linearization point for Op, we need to show that the history S_H obtained by ordering all operation calls in H by these $lin()$ points is valid. For that, we first prove the following auxiliary observations and claims. In the rest of this section, for ease of explanation, we call $lin(Op)$ the linearization point of Op.

The following observation says that, at any point, the size of each local queue a process maintains is $n+1$. This is used to show, in Claim 5.8, that if some process p reads its sequence number s from $Annc[q]$, for some q, then p keeps a copy of s in $annc\mathcal{Q}_p$ as long as $Annc[q] = s$. Moreover using this observation, Claim 5.9 proves that p executes n `getSeq()` calls between any two `getSeq()` calls that return the same sequence number $s \in I_p$.

Observation 5.7. *At any point and for any process p, each of the queues $used\mathcal{Q}_p$ and $annc\mathcal{Q}_p$ has size exactly $n+1$.*

Proof. Each of p's queues initially has $n+1$ elements with value $\perp$. These queues are only modified during a `getSeq()` call G by p. Operation call G contains only local steps and one atomic operation call. Hence all steps in this operation call are mapped to point $t_{G@inv} = t_{G@rsp}$. During G, process p enqueues exactly one element to $annc\mathcal{Q}_p$ in lines 4–7, and one element to $used\mathcal{Q}_p$ in line 11. Also p dequeues exactly one element from $annc\mathcal{Q}_p$ in line 8, and one from $used\mathcal{Q}_p$ in line 12. Thus, the size of each of p's queues is always $n+1$. □

Claim 5.8. *Consider a `getSeq()` call G by some process p. Suppose p reads $s \in I_p$ from $Annc[q]$ at $t_{G@3}$, and let t be the first point after t at which the value of $Annc[q] \neq s$, and $t = \infty$ if such a point does not exist in Λ. Then a copy of s remains in $annc\mathcal{Q}_p$ throughout $[t_{G@3}, t)$.*

Proof. Assume for the sake of contradiction that p dequeues the last copy of s from $annc\mathcal{Q}_p$ at some point $t' \in [t_{G@3}, t)$. Let t^* be the last point during $[t_{G@3}, t']$ at which p reads $Annc[q]$.

Since this happens at point $t_{G@3}$, such a point t^* exists in this interval. By the claim assumption, $Annc[q] = s$ throughout $[t_{G@3}, t)$, and thus

$$Annc[q] = s \text{ throughout } [t^*, t']. \tag{5.2}$$

Process p reads the entire array $Annc$ during any n consecutive `getSeq()` calls. Hence,

$$p \text{ completes fewer than } n \texttt{ getSeq() } \text{calls during } [t^*, t'], \tag{5.3}$$

since otherwise, t^* would not be the last time p reads $Annc[q]$ during this interval.

According to (5.2), process p reads sequence number s from $Annc[q]$ at t^*. Therefore at the same point, this process adds a copy of s to $annc\mathcal{Q}_p$ in lines 4–5. Process p dequeues exactly one element from $annc\mathcal{Q}_p$ during each `getSeq()` call, thus by Observation 5.7, once p enqueues a copy of s into $annc\mathcal{Q}_p$ at t^*, this copy remains in this queue until p completes at least another n `getSeq()` call after this point. Therefore by (5.3), this copy of s remains in $annc\mathcal{Q}_p$ throughout $[t^*, t']$. This contradicts the assumption that p dequeues the last copy of s from $annc\mathcal{Q}_p$ at t'. □

Claim 5.9. *Consider two* `getSeq()` *operation calls G_1 and G_2 by some process p, where G_1 is invoked before G_2. If both G_1 and G_2 return the same sequence number $s \in I_p$, then p completes at least n* `getSeq()` *calls between G_1 and G_2.*

Proof. Before process p returns sequence number s from G_1, it enqueues a copy of s into its $used\mathcal{Q}_p$ in line 11. In every `getSeq()` call only one element gets dequeued (in line 12). Hence by Observation 5.7, this copy of s remains in $used\mathcal{Q}_p$ during G_1, and the next n `getSeq()` calls that p executes after G_1. According to line 10, p chooses s as the return value of its `getSeq()` call again when no copy of s is stored in any of its queues. Thus, p completes at least n `getSeq()` calls after G_1 and before G_2. □

The following claim uses the results of Claims 5.8 and 5.9 to prove that Property ($\star$) holds.

Claim 5.10. *Consider a sequence number $s \in I_p$. Suppose $X = (x,s)$ at some point t, and $Annc[q] = s$ throughout $[t,t']$, for some $t' \geq t$, some process q, and some value x. Then, process p does not write (x',s) into X during $(t,t']$, for any value of x'.*

Proof. Since $X = (x,s)$ at t, p must have written this pair to X before t in a `DWrite(x)` call W. Let G be the `getSeq()` operation call that p completes before t in line 1 of W. Suppose for the sake of contradiction that p writes (x',s) to X during $(t,t']$ in line 2 of some `DWrite()` operation call W', for some value x'. Let G' be the `getSeq()` call that p completes before t' in line 1 of W'. Thus, both G and G' return s.

By Claim 5.9, p completes at least n `getSeq()` operation calls during $(t_{G@rsp}, t_{G'@inv})$. Let $G_1,\ldots,G_n$ be the first n `getSeq()` calls that p invokes during $(t_{G@rsp}, t_{G'@inv})$, in this order. Process p can invoke at most one `getSeq()` call during $(t_{G@rsp}, t]$, because otherwise the value of X would be overwritten before t. Hence, only G_1 can get invoked during $(t_{G@rsp}, t]$. Therefore,

$$\text{all } G_2,\ldots,G_n,G' \text{ complete during } (t,t']. \tag{5.4}$$

Process p increments its local variable ctr_p by 1 modulo n during each `getSeq()` call. Therefore, $ctr_p = q$ at the invocation of one `getSeq()` call $G'' \in \{G_2,\ldots,G_n,G'\}$. By the assumption, $Annc[q] = s$ throughout $[t,t']$. Thus by (5.4), p reads s from $Annc[q]$ in line 3 of G''. By Claim 5.8, a copy of s remain in $annc\mathcal{Q}_p$ throughout $[t_{G''@3}, t']$. Therefore, any `getSeq()` operation call by p that executes line 10 during this interval cannot return s. By (5.4), $t_{G'@10} \in [t_{G''@3}, t']$. Hence, G' does not return s, which is a contradiction. □

Recall that S_H is the sequential history that is obtained by ordering all operation calls in $H = \Gamma(\Lambda)$ by their linearization points. Let t_0 denote the point at which Λ starts. By our definition of $lin()$, a `DWrite(x)` operation call by some process p linearizes at the point p writes $(x,\cdot)$ to X. Let R_0 be the first `DRead()` call by some process q in Λ, and let R be any subsequent `DRead()` call by this process. Also let (x_0,g_0) and (x,g) be the return values of R_0 and R, respectively. To prove validity of S_H, it is enough to show that

(a) $X = (x,\cdot)$ and $X = (x_0,\cdot)$ at $lin(R)$ and $lin(R_0)$, respectively,

(b) $g_0 = \texttt{true}$, if and only if a `DWrite()` call linearizes during $[t_0, lin(R_0))$, and

(c) $g = \texttt{true}$, if and only if a `DWrite()` call linearizes during $[lin(R'), lin(R))$, where R' is the last `DRead()` call by q before R.

Because the first component of the return value of a `DRead()` call R is the same as the first component of the pair that the calling process reads from X in line 14 of R, Claim 5.11 in the following shows that Part (a) is true. Claim 5.12 shows that Part (b) holds. Claims 5.13 and 5.14 uses results of Claims 5.10 and 5.11 to show that Part (c) is also true. Therefore, S_H is a linearization of H.

Claim 5.11. *Suppose q reads some pair (x,s) from X in line 14 of a* `DRead()` *operation call R. Then $X = (x,s)$ at $lin(R)$.*

Proof. If $b_q = \texttt{true}$ at $t_{R@rsp}$, then $lin(R) = t_{R@14}$, and so the claim follows immediately. Now suppose $b_q = \texttt{false}$ at $t_{R@rsp}$, and so $lin(R) = t_{R@17}$. Thus, the if-condition in line 22 of R evaluates to true. This implies that q reads the same pair (x,s) from X in both line 14 and line 17 of R, and so the claim is true. □

Claim 5.12. *Let R be the first* `DRead()` *operation call by some process q in Λ. Then R returns $(\perp,\texttt{false})$, if no* `DWrite()` *call linearizes throughout $[t_0, lin(R))$, and otherwise, it returns $(\cdot,\texttt{true})$.*

Proof. Since a `DWrite()` linearizes when the calling process writes to X, we need to show that R returns $(\perp,\texttt{false})$, if no process writes to X throughout $[t_0, lin(R))$, and otherwise, it returns $(\cdot,\texttt{true})$. Operation call R is the first `DRead()` call by q, so

$$q \text{ reads } Annc[q] = \perp \text{ in line 15 of } R. \tag{5.5}$$

First suppose q reads $(\perp,\perp)$ from X in line 14 of R. This implies that no process writes to X throughout $[t_0, t_{R@14})$, because no process writes $\perp$ to the second component of X. Thus by

(5.5), the if-condition in line 18 evaluates true, and R returns $(\perp, b_q) = (\perp, \texttt{false})$. Unless q reads the same pair $(\perp, \perp)$ from X in line 17, then $lin(R) = t_{R@14}$, and so the claim is true for this case. If q reads $(\perp, \perp)$ from X in line 17, then no process writes to X throughout $[t_0, t_{R@17})$, and since $lin(R) = t_{R@17}$ in this case, the claim follows.

Next suppose that q reads some pair $(x, s) \neq (\perp, \perp)$ from X in line 14 of R. Hence, a process writes to X during $[t_0, t_{R@14})$, and so during interval $[t_0, lin(R))$, because $lin(R)$ is either $t_{R@14}$ or $t_{R@17}$. By (5.5), the if-condition in line 18 evaluates false, and R returns $(x, \texttt{true})$. Thus the claims follows. □

Claim 5.13. *Consider two consecutive* `DRead()` *operation calls* R_1 *and* R_2 *by some process* q*. Suppose* R_2 *returns some pair* $(\cdot, \texttt{false})$*. Then no* `DWrite()` *call linearizes throughout* $[lin(R_1), lin(R_2)]$.

Proof. Operation call R_2 returns $(\cdot, \texttt{false})$, therefore,

$$\text{the if-condition in line 18 of } R_2 \text{ evaluates to true, and } b_q = \texttt{false} \text{ at } t_{R_2@inv}. \tag{5.6}$$

Process q does not change the value stored in b_q during $[t_{R_1@rsp}, t_{R_2@inv}]$. Thus by (5.6),

$$b_q = \texttt{false} \text{ at } t_{R_1@rsp}. \tag{5.7}$$

Thus by definition of $lin()$, we have

$$lin(R_1) = t_{R_1@17}. \tag{5.8}$$

Let (x_1, s_1) and (x_2, s_2) be the pairs that process q reads from X in line 14 of R_1, respectively R_2. Since the value of $Annc[q]$ is only modified in line 16 of a `DRead()` operation call by q, $Annc[q] = s_1$ throughout $[t_{R_1@16}, t_{R_2@15}]$, and so q reads s_1 from this register in line 15 of R_2. Therefore by (5.6), $s_1 = s_2$.

By Claim 5.11, the value of the second component of X is $s_1 = s_2$ at both $lin(R_1)$ and $lin(R_2)$. Therefore, either no process writes to X throughout $[lin(R_1), lin(R_2)]$, and so the

claim follows, or the owner of s_1 writes this sequence number to the second component of X at some point during this interval. In the following, we show that the latter case is not possible.

Process q writes $s_1 = s_2$ to $Annc[q]$ at both $t_{R_1@16}$ and $t_{R_2@16}$, and since $Annc[q]$ is not changed elsewhere, we have

$$Annc[q] = s_1 = s_2 \text{ throughout } [t_{R_1@16}, t_{R_2@rsp}]. \tag{5.9}$$

According to (5.7), the if-condition in line 22 of R_1 evaluates to true. Thus, q reads the pair (x_1, s_1) from X at $t_{R_1@17}$. By (5.9), we have $Annc[q] = s_1$ throughout $[t_{R_1@17}, t_{R_2@rsp}]$, and so by Claim 5.10, the owner of sequence number s_1 does not write $(\cdot, s_1)$ to X during $[t_{R_1@17}, t_{R_2@rsp}]$, and so by (5.8) during $[lin(R_1), lin(R_2)]$. □

Claim 5.14. *Consider two consecutive* `DRead()` *operation calls* R_1 *and* R_2 *by some process* q*. Suppose* R_2 *returns some pair* $(\cdot, \texttt{true})$*. Then a* `DWrite()` *call linearizes during* $[lin(R_1), lin(R_2)]$.

Proof. Operation R_2 returns $(\cdot, \texttt{true})$, therefore either the if-condition in line 18 of R_2 evaluates to true and $b_q = \texttt{true}$ at $t_{R_2@inv}$, or the same if-condition evaluates to false.

Suppose that the former happens. Process q's local variable b_q does not change during $[t_{R_1@rsp}, t_{R_2@inv}]$. Thus, $b_q = \texttt{true}$ at $t_{R_1@rsp}$, and so $lin(R_1) = t_{R_1@14}$. This also implies that the if-condition in line 22 of R_1 evaluates to false. Hence, q reads different pairs from X in line 14 and line 17 of R_1. Therefore, a process writes to register X and so a `DWrite()` call linearizes during $[t_{R_1@14}, t_{R_1@17}] \subseteq [lin(R_1), lin(R_2)]$.

Next suppose the latter happens, i.e. the if-condition in line 18 of R_2 evaluates to false and so R_2 returns in line 21. Let (x_1, s_1) and (x_2, s_2) be the pairs q reads from X in line 14 of R_1, respectively R_2. Register $Annc[q]$ can only be modified by q and only in line 16 of a `DRead()` operation call, so $Annc[q] = s_1$ throughout $(t_{R_1@16}, t_{R_2@15}]$, and so q reads s_1 from this register in line 15 of R_2. Since the if-condition in line 18 of R_2 evaluates to false, $s_1 \neq s_2$. By Claim 5.11, $X = (x_1, s_1)$ at $lin(R_1)$, but $X = (x_2, s_2) \neq (x_1, s_1)$ when q reads this

```
shared:
LL/SC X = ◇

/* ⊥ is the initial value of the
   ABA-detecting register and ◇ is a
   value that no process writes to this
   register.                          */

Operation DWrite_p(x)
27 X.LL()
28 X.SC(x)

local (to each process p):
old_p

Operation DRead_p()
29 if X.VL() then
30     return (old_p,false)
31 old_p := X.LL()
32 if old_p = ◇ then
33     old_p = ⊥
34     return (old_p,false)
35 return (old_p,true)
```

Figure 5.2: Implementation of an ABA-detecting Register from LL/SC/VL.

register at $t_{R_2@14}$. Thus, some process writes to X and so a `DWrite()` call linearizes during $[lin(R_1), t_{R_2@14}] \subseteq [lin(R_1), lin(R_2)]$. □

5.3 ABA-Detecting Register from a Single LL/SC/VL

This section provides a simple implementation of an ABA-detecting register from a single LL/SC/VL object.

5.3.1 Algorithm Description

This algorithm uses one LL/SC/VL object X as depicted in Figure 5.2. Let D be the set of all possible values that a process can write to the implemented ABA-detecting register through a `DWrite()` call, and let $\perp \in D$ be the initial value of this ABA-detecting register. We initialize object X with a value $\diamond$ that is not in D, to identify whether any `DWrite()` has linearized before any process' first `DRead()` call. Each process p maintains a local variable old_p that stores the return value of its last `DRead()` call.

During a `DWrite(x)` call W, the calling process executes an X.`LL()` call followed by an X.`SC(s)` call (lines 27 and 28). If this `SC()` call succeeds, then the value of X changes to x, and W linearizes in line 28. Otherwise, another process must have executed a successful X.`SC()` call between lines 27–28 of W. Then W linearizes just before that successful `SC()` call.

During each DRead() call R, the calling process p checks if it has a valid link to X by executing an X.VL() (line 29). If p's X.VL() call in line 29 returns true, then R is not p's first DRead() operation call, and no successful X.SC() call has happened since p's last X.LL() call during an earlier DRead() operation call. In this case, p just returns the same value as the one that was returned in its last DRead() call. This value is stored in old_p. That DRead() call linearizes at the VL() call in line 29.

If p's X.VL() call (in line 29) returns false, then p does not hold a valid link to X, so it initiates a new link and updates its local variable old_p by executing an X.LL() (line 31). If no DWrite() call has linearized before this point, then p reads $\diamond$ which is the value initially stored in X. Therefore in lines 32–34, p updates old_p and returns the initial value $\perp$ of the ABA-detecting register and false. Otherwise, p has to return the value it loads from X in line 31 paired with true, to indicate that the value of the ABA-detecting register has changed since its last DRead() call. In both cases, that DRead() call linearizes at the LL() call executed in line 31.

5.3.2 Proof of Theorem 5.3

The implementation of an ABA-detecting register provided in Figure 5.2 uses only one LL/SC/VL object, and each DWrite() and DRead() operation call executes only at most two operations on the LL/SC/VL object. Hence to prove Theorem 5.3, it remains to show that the implementation in Figure 5.2 is linearizable.

Consider a transcript Λ obtained from executing DWrite() and DRead() operations on the implemented ABA-detecting register, and let $H = \Gamma(\Lambda)$. Also let t_0 denote the point at which Λ starts. As discussed in the algorithm description, we define a linearization point $lin(Op)$ for each operation call $Op \in \{$DWrite(),DRead()$\}$ in H: A DWrite() operation call W with a successful SC() call in line 28 linearizes with that successful SC() call, i.e. $lin(W) = t_{W@28}$. If the SC() call during W fails, then $lin(W)$ is the point immediately before the first successful SC() that gets executed after t_{W27}. Such a successful SC() must occur during $[t_{w@27}, t_{W@28}]$, because p's SC() call in line 28 fails. For a DRead() operation call R, $lin(R) = t_{R@29}$ if it returns in line 30,

and otherwise $lin(R) = t_{R@31}$.

The linearization point of each operation call is between the invocation and the response of that operation call. Hence, it is enough to show that the sequential history S_H obtained by ordering operation calls in H by their linearization points is valid.

Every `DWrite()` operation call linearizes either at or immediately before the point of some successful X.`SC()` call. Therefore, at any point the value of X is equal to the value of the `DWrite()` operation call that linearized last. Sequential history S_H is valid, if every `DRead()` operation call R in S_H by some process p which returns some pair (x,g), satisfies the following.

(a) If there is no `DWrite()` call before R in S_H, then $x = \perp$ and $g = \texttt{false}$

(b) Otherwise,

(b1) $X = x$ at $lin(R)$, and

(b2) $g = \texttt{true}$ if R is the first `DRead()` call by p, and (otherwise)

(b3) $g = \texttt{true}$ if and only if a `DWrite()` call linearizes between the linearization point of the last preceding `DRead()` call by p and $lin(R)$.

Fix a `DRead()` operation call R by p that returns some pair (x,g), and let S'_H be the prefix of S_H which ends just before the invocation event of R. We show that if S'_H is valid, then (S'_H,R) is also valid.

Proof of Part (a). If no `DWrite()` call linearizes before the linearization point of R, then no process executes a successful X.`SC()` call and so $X = \diamond$ throughout $[t_0,lin(R)]$. First assume that R is the first `DRead()` call by p in S_H. This implies that p does not hold a valid link to X, hence, the if-condition in line 29 of R fails. Therefore, $lin(R) = t_{R@31}$. Since $X = \diamond$ throughout $[t_0,lin(R)]$, p's if-condition in line 32 evaluates to true, and p sets $old_p = \perp$ and returns $(\perp,\texttt{false})$ which proves Part (a). Next, assume that R is not the first `DRead()` call by p in S_H. Since there is no successful X.`SC()` call throughout $[t_0,lin(R)]$, p has a valid link since

the linearization point of its first DRead() call. Moreover, p does not change the value of old_p after it writes $\perp$ to this variable during its first DRead() call. Therefore, p returns $old_p = \perp$ paired with false in line 30.

Proof of Part (b). First suppose that R is the first DRead() call by p. Therefore, p does not hold a valid link to X when it executes line 29 of R, and so the if-condition in this line fails. Thus, p loads the current value of X into old_p in line 31. Since a DWrite() has linearized before $lin(R) = t_{R@31}$, $old_p \neq \diamond$, and so p returns the value it loads from X at $lin(R)$ paired with true. This proves Parts (b1) and (b2) for this case.

Next assume that R is not the first DRead() call by p. Let R' be the last DRead() call that p executes before R, in which p executes an X.LL() call in line 31. (R' may not be the last preceding DRead() call by p before R.) Such a DRead() call exists because p always executes line 31 in its first DRead() call, and R is not the first DRead() call by p. Process p only changes the value of old_p during a DRead() which executes an X.LL() call in line 31, so

$$\text{the value of } old_p \text{ does not change throughout } (lin(R'), lin(R)). \tag{5.10}$$

First suppose $g = \texttt{false}$. Since a DWrite() linearizes before R, p cannot load $\diamond$ from X in line 31 of R. So R must return in line 30, and $lin(R) = t_{R@29}$. Thus p has a valid link to X at $lin(R)$. This implies that this link has not changed since $lin(R')$ and so no successful X.SC() has been executed throughout $[lin(R'), lin(R)]$. Thus, no DWrite() operation call linearizes throughout $[lin(R'), lin(R)]$, and so between the linearization point of the last preceding DRead() call before R in S_H and $lin(R)$, which proves Part (b3), if $g = \texttt{false}$. Since no DWrite() linearizes between R' and R, but a DWrite() linearizes before R, it must have linearized before R' as well. Therefore, by induction hypothesis Part (b1), and because old_p at $lin(R')$ is the return value of R', we have

$$X = old_p \text{ at } lin(R') = t_{R'@31}. \tag{5.11}$$

The value of X does not change during $[lin(R'), lin(R)]$. Hence, Part (b1) for this case follows

from (5.10) and (5.11).

Next suppose that $g = \texttt{true}$. Then R returns in line 35, and $lin(R) = t_{R@31}$. Part (b1) immediately follows because $x = old_p$ is the value that p loads from X at $lin(R)$. Moreover, p's X.VL() call in line 29 must have returned false, therefore an X.SC() call must have been executed during $[lin(R'), t_{R@29}] \subseteq [lin(R'), lin(R)]$. Therefore, a DWrite() must have linearized between the linearization point of the last preceding DRead() call by p in S_H and $lin(R)$, which proves Part (b3) for this case. This concludes the proof of Theorem 5.3.

5.4 LL/SC/VL from a Single Bounded CAS

This section presents a wait-free implementation of LL/SC/VL from a single bounded CAS object. The implementation has $O(n)$ step complexity, and thus, by Corollary 5.4, is optimal with respect to time-space tradeoff. The pseudo-code is presented in Figure 5.3 and correctness proofs can be found in Section 5.4.2.

5.4.1 Algorithm Description

This implementation uses only one CAS object X. This object stores a pair (x, a), where x represents the value of the implemented LL/SC/VL object, and a is an n-bit string. The p-th bit of a, a_p, is used to indicate whether p holds a valid link to the LL/SC/VL object. More specifically,

$$\begin{aligned} &a_p = 1, \text{ if and only if an SC() operation call has linearized} \\ &\text{since the linearization point of } p\text{'s last LL() call.} \end{aligned} \tag{5.12}$$

To maintain (5.12), during an SC() call, in addition to change the value of x, process p tries to set all bits of a to 1, and during an LL() call, p tries to reset a_p. This is done by executing an X.CAS() call. However, this call may fail, because some other process q executes a successful X.CAS() call to either change the values of x and a during a successful SC() call, or to reset a_q during an LL() call. The idea is that p makes at most n attempts until its CAS() succeeds. If p's

```
shared:
  CAS X = (⊥, 2^n − 1)
  // a_p represents the p-th bit of the second component of X.

Operation SC_p(x)
36 for i := 1 to n do
37     (y, a) := X.Read()
38     if a_p = 1 then                                  // if p's link is not valid
39         return false
40     if X.CAS((y, a), (x, 2^n − 1)) then     // try to change the value and set all bits
41         return true
   // a successful SC() has linearized since the beginning of this operation call
42 return false

Operation VL_p()
43 (x, a) := X.Read()
44 if a_p = 0 then                                      // if p's link is valid
45     return true
46 else
47     return false

Operation LL_p()
48 (x, a) := X.Read()
49 if a_p = 0 then  return x ;                          // if p's link is valid
50 for i := 1 to n do
51     (x', a') := X.Read()
52     if X.CAS((x', a'), (x', a' − 2^p)) then          // try to reset p's bit
53         return x'
   // a successful SC() has linearized since line 48
54 return x
```

Figure 5.3: An LL/SC/VL Implementation from Bounded CAS.

CAS() fails n times, then X must have changed n times while p's attempts fail. Since a process only changes X during an LL() call if its bit is not already 0, at most $n-1$ of those changes of X can be due to a successful CAS() executed in an LL() operation call. Therefore, at least one of those changes is because of a successful CAS() on X executed in an SC() operation. Thus, p can linearize just before that successful SC() call linearizes, and ensure that (5.12) is satisfied.

With this idea, our implementation is as follows. During an SC(x) call, if p reads 1 from a_p, it returns false immediately, and linearizes at this read. Otherwise, p makes at most n attempts

to write $(x,2^n-1)$ into X by executing a CAS() call. With the first successful CAS() call, p's operation linearizes, and p returns true. If p's CAS() fails n times, then a successful SC() linearizes while p makes those n failed attempts. Thus, p's operation linearizes just before that successful SC() call and returns false.

Similarly, during a LL() call, p reads X. If it reads 0 from a_p and x from the first component of X during an LL() call, it returns x, and linearizes at this read. Otherwise, it executes at most n X.CAS() calls to reset a_p, and with the first successful attempt, its operation linearizes, and p returns the last value it read from the first component of X. If p fails all n times, then p returns the first value it read from X during its current LL() call, and linearizes at that X.Read(). Moreover, p's bit, which is now set because of that successful SC() call, indicates that p does not hold a valid link.

Operation VL() is simple: process p checks whether a_p is set, and if yes, it returns false, otherwise it returns true. This operation linearizes with the only read of X.

5.4.2 Proof of Theorem 5.5

Consider the LL/SC/VL object implemented form a single CAS object as given in Figure 5.3. Each operation requires at most $O(n)$ CAS() operation calls. So to prove Theorem 5.5, it is enough to show that this implementation is linearizable. In the rest of this section, we call p-th bit of the second component of X as p's bit.

Let Λ be a transcript that is obtained by executing LL(), SC(), and VL() operation calls on the LL/SC/VL object of Figure 5.3. Also let t_0 denote the point at which Λ starts. We define a linearization point $lin(Op)$ for each operation call $Op \in \{\texttt{LL()},\texttt{SC()},\texttt{VL()}\}$ in $H = \Gamma(\Lambda)$, as follows. For an SC() operation call S, we define $lin(S) = t_{S@rsp}$. Hence, for a successful SC() operation call S, $lin(S)$ is the point at which its CAS() in line 40 succeeds. For a VL() operation call V, we define $lin(V) = t_{V@inv} = t_{V@rsp}$. For an LL() operation call L, if L returns in either line 49 or line 54, we let $lin(L)$ be the point at which the calling process reads X in line 48 of L, so $lin(L) = t_{L@inv}$. If L returns in line 53, then $lin(L) = t_{L@rsp}$ is the point at which its CAS()

call in line 52 succeeds. The linearization point of each operation call is between its invocation and its response. So it only remains to show that the sequential history S_H obtained by ordering operation calls in H by their linearization points is valid.

The first component of the pair stored in X only changes to some value x at the linearization point of a successful `SC(`x`)` call. Thus to prove that S_H is valid, we need to show that

(a) if an `LL()` call L returns x, then $X = (x, \cdot)$ at $lin(L)$,

(b) an `SC()` call S succeeds, if and only if there is no successful `SC()` call that linearizes during $[lin(L), lin(S)]$, where L is the last `LL()` call by the same process before S (and S fails, if there is no such operation call L), and

(c) a `VL()` call V returns true, if and only if there is no successful `SC()` call that linearizes during $[lin(L), lin(V)]$, where L is the last `LL()` call by the same process before V (and V returns false, if there is no such operation call L).

By definition of the linearization point for an `LL()` operation call, (a) follows immediately. All process's bits in X are initially 1, and only a `LL()` by some process p can change p's bit to 0. Hence, if an `SC()` or `VL()` call is not preceded by an `LL()` call by the same process in Λ, then by implementation, this operation call correctly returns false in line 39 or line 47, respectively. Claim 5.17 in the following proves (b), for an `SC()` call that is preceded by an `LL()` operation call by the same process. Similarly, Claim 5.18 proves (c), for a `VL()` call that is preceded by an `LL()` operation call by the same process. These two claims use the result of Claims 5.15 and 5.16 that are discussed first.

Claim 5.15. *Consider some process p that executes the for-loop in lines 36–41 of an* `SC()` *or lines 50–53 of an* `LL()` *call. If all n* `CAS()` *calls during this for loop fail, then a successful* `SC()` *call linearizes while p is executing this for-loop.*

Proof. Consider a for-loop executed in lines 36–41 of an `SC()`, respectively lines 50–53 of an `LL()`, call by p. Let I denote the interval that starts and ends with this for-loop. Also let

$C_1,\ldots,C_n$ be the `CAS()` calls that p executes in this for-loop (in line 40, respectively 52), and let R_i be the `Read()` operation call p executes just before C_i (in line 37, respectively 51), for $i \in \{1,\ldots,n\}$. Operation C_i fails if and only if a process executes a successful `CAS()` operation call in line 40 or 52, between p's R_i and C_i. Since all $C_1,\ldots,C_n$ fail,

$$\text{at least } n \text{ CAS() operation calls executed in line 40 or 52 succeed during } I. \tag{5.13}$$

A process q only executes a `CAS()` operation call in line 52, if its bit in X is 1, and if this `CAS()` call succeeds, it resets q's bit in the second component of X to 0. Hence, each of these n bits can change to 0 at most once, before some `CAS()` call in line 40 by some process succeeds to change that bit to 1. Since none of p's `CAS()` operation calls during I succeeds, at most $n-1$ successful `CAS()` calls are executed in line 52 during I. Therefore by (5.13), at least one `CAS()` call that is executed in line 40 of an `SC()` call succeeds during I. Thus, one `SC()` call linearizes during this interval. □

Claim 5.16. *Consider some* `LL()` *operation call L by some process p. Process p's bit in X is set at $t_{L@rsp}$, if and only if some successful* `SC()` *operation call linearizes during $[lin(L), t_{L@rsp}]$.*

Proof. Suppose p's bit in X is set at $t_{L@rsp}$. If L returns in line 49, then p's bit in X is not set when p reads X in line 48 at $lin(L)$. This bit is only set when a `CAS()` call succeeds during an `SC()` operation call. Hence, a successful `SC()` operation call linearizes during $[lin(L), t_{L@rsp}]$. If L returns in line 54, then all n `CAS()` calls that p executes during its for-loop in lines 50–53 fail. Therefore by Claim 5.15, a successful `SC()` operation call linearizes while p is executing its for-loop. Since $lin(L) = t_{L@48}$, this successful `SC()` call linearizes during $[lin(L), t_{L@rsp}]$. It is not possible that L returns in line 53, because this implies that p's last `CAS()` call in line 52 of L must have been successful, and so p's bit in X must have changed to 0 at $lin(L) = t_{L@rsp}$.

Next, we show that if p's bit in X is 0 at $t_{L@rsp}$, then

$$p\text{'s bit in } X \text{ is 0 throughout } [lin(L), t_{L@rsp}]. \tag{5.14}$$

This implies that no successful SC() linearizes throughout $[lin(L), t_{L@rsp}]$, because all processes' bits in X change to 1 when a CAS() call in line 40 of an SC() operation call succeeds at the linearization point of that SC() call.

To prove (5.14), first suppose L returns in line 49. Therefore, p's bit is 0 when p reads X in line 48 at $lin(L)$. This bit can only be changed to 0 by p and only in line 52, which is not executed during L in this case. Hence, (5.14) follows. If L returns in line 53, p's last CAS() call in line 52 of L must have succeeded to change p's bit to 0 at $lin(L) = t_{L@rsp}$. So (5.14) is true. It is not possible that L returns in line 54 if p's bit in X is 0 at $t_{L@rsp}$, because this only happens if p's bit in X is 1 when p reads X in line 48 of L, and all n CAS() calls that p executes to change this bit to 0 in line 52 also fail. □

Claim 5.17. *Consider an* SC() *operation call S by some process p, and let L be the last preceding* LL() *call by p. Operation call S is successful, if and only if no successful* SC() *call linearizes during $[lin(L), lin(S)]$.*

Proof. First suppose that S is successful, and so it returns in line 41. This implies that an X.CAS() operation call C executed in line 40 of S succeeds at $lin(S)$. Let t be the last time p reads X in line 37 of S before C. Process p reads value 0 from its bit when it reads X at t. The value of this bit does not change during $[t, lin(S)]$, because C is successful. Moreover, p's bit can only change to 0 in line 52 of an LL() operation call by p, hence,

$$p\text{'s bit in } X \text{ is 0 throughout } [t_{L@rsp}, lin(S)]. \tag{5.15}$$

Therefore, no successful SC() linearizes throughout $[t_{L@rsp}, lin(S)]$, as otherwise the value of p's bit would change to 1. Finally, by (5.15) and Claim 5.16, no successful SC() operation call linearizes during $[lin(L), t_{L@rsp}]$. Hence, no successful SC() linearizes throughout $[lin(L), lin(S)]$.

Next suppose that S is not successful. First consider the case that S returns in line 39. Let t be the last time p reads X in line 37 before $lin(S) = t_{S@rsp}$. Hence, p reads 1 from its bit in X at t. Depending on the value of p's bit in X at $t_{L@rsp} < t$, there are two cases: If p's

bit is 0 at $t_{L@rsp}$, then some process sets this bit with a successful `CAS()` at the linearization point of a successful `SC()` operation call during $(t_{L@rsp}, t] \subseteq [lin(L), lin(S)]$. Otherwise, if p's bit is 1 at $t_{L@rsp}$, then by Claim 5.16, a successful `SC()` operation call linearizes during $[lin(L), t_{L@rsp}] \subseteq [lin(L), lin(S)]$.

The final case is when S returns in line 42. This implies that all n `CAS()` operation calls of p during S fail. Thus by Claim 5.15, a successful `SC()` operation call linearizes while p executes this for-loop, and so during $[lin(L), lin(S)]$, because $lin(S) = t_{S@rsp}$. □

Claim 5.18. *Consider a* `VL()` *operation call V by some process p, and let L be the last preceding* `LL()` *call by p. Operation call V returns true, if and only if no successful* `SC()` *call linearizes during $[lin(L), lin(V)]$.*

Proof. First suppose that V returns true. Hence, p reads value 0 from its bit in X in line 43 of V at $lin(V)$. This bit can only be reset in line 52 of an `LL()` operation call by p, hence,

$$p\text{'s bit is 0 throughout } [t_{L@rsp}, lin(V)]. \tag{5.16}$$

Therefore, no successful `SC()` linearizes throughout $[t_{L@rsp}, lin(V)]$, as otherwise the value of p's bit in X would change to 1. Moreover by (5.16) and Claim 5.16, no successful `SC()` operation call linearizes during $[lin(L), t_{L@rsp}]$. Therefore, no successful `SC()` linearizes throughout $[lin(L), lin(V)]$.

Next suppose that V returns false. Therefore, p's bit in X is 1 when p reads X in line 43 of V at $lin(V)$. If p's bit is 0 at $t_{L@rsp}$, then some process sets this bit with a successful `CAS()` at the linearization point of a successful `SC()` operation call during $(t_{L@rsp}, lin(V)] \subseteq [lin(L), lin(V)]$. If p's bit is 1 at $t_{L@rsp}$, then by Claim 5.16, a successful `SC()` operation call linearizes during $[lin(L), t_{L@rsp}] = [lin(L), lin(V)]$. □

Chapter 6

More Efficient Long-Lived Test-And-Set

This chapter provides several space- and time-efficient implementations of randomized long-lived test-and-set (TAS) objects from registers. These results are obtained from general transformations of randomized one-time TAS implementations e.g. the ones by Alistarh and Aspnes (2011), Giakkoupis, Helmi, Higham and Woelfel (2013, 2015), and Giakkoupis and Woelfel (2012) into long-lived TAS implementations.

6.1 Results

We present three transformations of one-time TAS objects into long-lived ones. Figure 6.1 previews existing long-lived TAS implementations, and some examples of the ones that we can achieve with transformations introduced in this chapter.

Our base construction, in Section 6.2, transforms any one-time TAS object implemented from m registers into a long-lived one that uses $O(n+m)$ registers while preserving the asymptotic step complexity of `TAS()`, and providing an implementation of `Reset()` with $O(m)$ worst-case step complexity. This transformation is deterministic, so the resulting long-lived TAS implementation is randomized if and only if the original one-time TAS is randomized.

Theorem 6.1. *Any one-time TAS object O implemented from m registers, each of size b bits, can be transformed into a long-lived TAS object L with the following properties:*

- *L uses $O(n+m)$ registers, each of size $\max\{\lceil\log(2n+m+2)\rceil, b\}$ bits,*
- *L.`TAS()` has asymptotically the same (expected) step complexity as O.`TAS()`, and*
- *L.`Reset()` has worst-case step complexity $O(m)$.*

One-Time TAS				Long-Lived TAS					
Reference	Step Complexity of `TAS()`	Num. of Registers	Size of Registers	Applying Transformation	Step Complexity of `TAS()`	Step Complexity of `Reset()`	Num. of Registers	Size of Registers	Adversary
Afek et al. (1992)	$O(\log n)$ exp.	$O(n)$	$\Theta(\log n)$	Afek et al. (1992)	$O(n)$ exp.	$O(1)$	$O(n)$	$\Theta(n)$	Adaptive
				Hoepman (1999)	$O(\log n)$ exp.	$O(n)$	$O(n^2)$	$\Theta(\log n)$	Adaptive
Giakkoupis et al. (2015)	$O(\log^* n)$ exp.	$O(\log n)$	$\Theta(\log n)$	Theorem 4.1 and Lemma 4.2	$O(\log^* n)$ exp.	$O(1)$	$O(n^2 \log n)$	$\Theta(\log n)$	Oblivious
				Corollary 6.2	$O(\log^* n)$ exp.	$O(\log n)$ exp.	$O(n)$	$\Theta(\log n)$	
				Corollary 6.5	$O(\log^* n)$ exp.	$O(1)$	$O(n \log n)$	$\Theta(\log n)$	
Giakkoupis and Woelfel (2012)	$O(\log \log n)$ exp.	$O(n)$	$\Theta(\log n)$	Theorem 6.3	$O(\log \log n)$ exp.	$O(\log \log n)$ exp.	$O(n)$	$\Theta(\log n)$	Oblivious

Figure 6.1: Results on Randomized Long-Lived Test-And-Set

Giakkoupis, Helmi, Higham and Woelfel (2015) present a randomized implementation of a one-time TAS object with $O(\log^* n)$ expected step complexity, that uses $\Theta(\log n)$ registers, each of size $O(\log n)$. Applying the transformation of Theorem 6.1 to this object yields:

Corollary 6.2. *There is an implementation of a long-lived TAS from $O(n)$ registers, each of size $\Theta(\log n)$ bits, where* `TAS()` *has $O(\log^* n)$ expected step complexity, and* `Reset()` *has worst-case step complexity $O(\log n)$ against the oblivious adversary.*

The space complexity of $O(n)$ is *optimal* based on the lower bound on the space requirement of mutual exclusion (Burns and Lynch, 1993). While `Reset()` calls of the long-lived TAS implementation of Corollary 6.2 are significantly slower than `TAS()` calls, this may still be reasonable in applications where `Reset()` operations may be executed less frequently than `TAS()` operations.

Our second transformation, in Section 6.3.1, uses the base construction with some additional ideas that are particularly tailored to a one-time TAS implementation by Giakkoupis and Woelfel (2012), in which `TAS()` calls require $O(\log\log n)$ steps in expectation. This enables us to reduce the step complexity of `Reset()` to $O(\log\log n)$ in expectation, while maintaining the optimal $O(n)$ space requirement. However, the step complexity bound of `TAS()` is increased to $O(\log\log n)$ in expectation compared to that achieved in Corollary 6.2.

Theorem 6.3. *There is an implementation of a long-lived TAS from $O(n)$ registers, each of size $\Theta(\log n)$ bits, such that* `TAS()` *and* `Reset()` *have $O(\log\log n)$ expected step complexity against the oblivious adversary.*[1]

These are the first implementations of a long-lived TAS object with $O(n)$ space complexity and sub-linear step complexity for both `TAS()` and `Reset()`.

In our third transformation in Section 6.3.2, we combine the idea of long-lived TAS by Hoepman (1999) with our recycling technique as well as the technique from Section 4.5, which allows a process to sequentially reset multiple registers in $O(1)$ steps. The resulting implementation has

[1] *The step complexity bound even holds for the stronger read-write oblivious adversary model used in (Giakkoupis and Woelfel, 2012).*

constant step complexity for `Reset()`, and preserves the asymptotic step complexity of `TAS()` calls. However, we lose the space optimality by requiring $\Theta(n \cdot m)$ registers, where m is the number of registers used to implement the one-time TAS object.

Theorem 6.4. *Any one-time TAS object O implemented from m registers, each of size $O(\log n)$ bits, can be transformed into a long-lived TAS object L with the following properties:*

- *L uses $O(n \cdot m)$ registers, each of size $O(\log n)$ bits,*
- *L.`TAS()` has, up to a constant additive term, the same (expected) step complexity as O.`TAS()`, and*
- *L.`Reset()` has constant worst-case step complexity.*

Currently, the most space-efficient published randomized wait-free one-time TAS implementation uses $O(\log n)$ registers, each of size $O(\log n)$, and has step complexity $O(\log^* n)$ (Giakkoupis, Helmi, Higham and Woelfel, 2015). Applying Theorem 6.4 to this construction yields the following.

Corollary 6.5. *There is an implementation of a long-lived TAS from $O(n \log n)$ registers, each of size $\Theta(\log n)$ bits, such that `TAS()` has $O(\log^* n)$ expected step complexity against the oblivious adversary, and `Reset()` requires $O(1)$ steps in the worst case.*

On the other hand, a lower bound by Styer and Peterson (1989) shows that any deadlock-free one-time TAS object requires at least $\Omega(\log n)$ registers, so the space complexity of a deadlock-free long-lived TAS object obtained from a direct application of Theorem 6.4 cannot go below $\Omega(n \log n)$.

6.2 Base Algorithm

This section describes our basic transformation of any one-time TAS object O, implemented from registers, into a long-lived TAS object L. Later sections of this chapter provide other constructions that build on the techniques introduced here.

6.2.1 High Level Idea

Our construction of L from O is simplified by exploiting two properties of long-lived test-and-set.

(I) Any L.`TAS()` that overlaps with an L.`Reset()` can safely return 1, and be linearized before that L.`Reset()`.

Of course the construction must ensure that such an L.`TAS()` does not interfere with the overlapping L.`Reset()`. Also, no two L.`Reset()` operations can overlap, because L.`Reset()` is only invoked by a process that has won an L.`TAS()` operation and has not subsequently executed an L.`Reset()`.

(II) Once a L.`Reset()` has been invoked there cannot be another invocation of L.`Reset()` without an intervening L.`TAS()` that wins.

Let R_0 to R_{m-1} be the registers that implement O. Also let I_B be a set of indices. Our construct uses a pool of registers, B, of size I_B, and an array, Ptr, of size m, where each entry $Ptr[i]$ is an index in I_B. Register $B[b]$ is in use whenever $Ptr[i] = b$, for some $i \in \{0,\ldots,m-1\}$.

If the entire Ptr array could be updated atomically and $|I_B|$ were infinite, then a solution would be immediate: Each L.`Reset()` would atomically change each entry $Ptr[i]$, for $i \in \{0,\ldots,m-1\}$, to an index f, such that $B[f]$ has never been in use. The implementation of L.`TAS()` is the same as O.`TAS()`, except that each read or write of register R_i is implemented by the same operation on the register $B[Ptr[i]]$. By properties (I) and (II), L.`TAS()` calls executing an operation on a register that is not currently in use, could safely return 1 and halt; those executing on registers that are in use would compete to win or lose.

Linearizable Reset. Our first goal is to maintain this correctness while removing the atomicity of the `Reset()`. The construction employs a single bit ABA-detecting register C. An L.`Reset()` operation call begins with a C.`DWrite(1)` call. It then, one at a time, writes indices of registers in B that have never been in use into the Ptr array. Finally it sets C back to 0 with a C.`DWrite(0)`

call. Each L.TAS() operation T is modified to execute a C.DRead() at the beginning, and after each read or write on Ptr or on B. If the first C.DRead() returns $(1,\cdot)$, then T overlaps with an L.Reset(). Otherwise, if a subsequent C.DRead() returns $(\cdot,\texttt{true})$, then an L.Reset() operation has begun since T was invoked. In either case T returns 1 and halts. As long as neither of these conditions cause T to lose, it continues to compete to win on the current set of registers from B as indicated in the Ptr array.

Bounding the Space. Now we describe how to reduce the infinite pool of registers, B, to only a small finite pool of registers. Various memory reclamation techniques (Braginsky, Kogan and Petrank, 2013; Gidenstam, Papatriantafilou, Sundell and Tsigas, 2009; Herlihy, Luchangco and Moir, 2002; Michael, 2004b) can be used to bound the space. However, if they are used directly without any additional techniques, they increase the step complexity or they may even break the wait-freedom of the implementation. Also, for most memory reclamation techniques, such as the ones by Herlihy et al. (2002), Gidenstam et al. (2009), and Braginsky et al. (2013), stronger primitives such as CAS and FAA are required. Here we use a technique that is time-efficient and uses only registers.

Our idea for memory reclamation builds on the recycling technique of Chapter 4, but we modify that technique for this specific application to reduce the required space. Because no two L.Reset() calls overlap, all sequential data structures used for recycling can be shared among different processes (as opposed to being local to each process), because they will be accessed only sequentially. The resulting long-lived implementation only requires $O(n+m)$ more registers than the one-time TAS implementation (as opposed to $O(n^2+m)$, for the more general technique in Chapter 4).

Our implementation now uses array B of size $|I_B| = 2n+m+1$, and an announce array $Annc$ of size n. Each process p announces the index of an entry of B that it is about to access (read or write) during an L.TAS() call, by writing the index into $Annc[p]$. Moreover, p also executes a C.DRead() after each announcement, and if it receives $(\cdot,\texttt{true})$ then it returns 1

immediately. During an L.`Reset()`, the calling process replaces the index stored in each entry $Ptr[i]$, $i \in \{0,\ldots,m-1\}$, with an index $f \in I_B$, such that $B[f]$ is currently not in use and is not announced.

We show that any step that a process p takes during an L.`TAS()` call T after the invocation of an overlapping L.`Reset()` call R is not going to affect the remaining operation calls: The only case that p can erroneously affect the object is if p writes to some register $B[f]$ after the resetter puts this register into use during R. Now, every second step by p in T is a C.`DRead()` call. Thus, p executes at most one other operation after the invocation of R, and before it sees the change in C and returns 1. Therefore, p writes to $B[f]$ only if it announces index f before R is invoked. However, the resetter would see p's announcement of f, and it would not put $B[f]$ into use again before p announces another object. Thus, it is not possible that p writes to $B[f]$ after the resetter puts this register into use during R.

Figure 6.2 shows the implementation of the `TAS()` and `Reset()` operations of long-lived implementation L of TAS. Helper function `recycle()` is called from an L.`Reset()` operation call and it provides the memory reclamation functionality of this implementation, as described in more details in the following. The pseudocode for the `recycle()` implementation is provided in Figure 6.3. The correctness proof of the base algorithm including the memory management part is provided in Section 6.4.

6.2.2 Detailed Description of recycle()

The memory management of this implementation is mainly done in the `recycle()` operation. This operation takes an index that is about to be replaced with another index in an entry of Ptr, and returns the index of a register that is currently not in use and is not announced.

In a naive implementation of the `recycle()` operation, processes would read the entire $Annc$ and Ptr arrays in each `recycle()` call. In order to achieve constant step complexity for `recycle()`, processes distribute the work of scanning $Annc$ over n `recycle()` calls similar to the deamortization technique of Chapter 4, and avoid scanning Ptr altogether, as described below.

```
shared
Array<register> B[0...2n+m] = (⊥,...,⊥)
Array<int> Ptr[0...m-1] = (0,...,m-1)
Array<int> Annc[0...n-1] = (⊥,...,⊥)
ABA-detecting register C = 0

Operation Reset_p()
1 C.DWrite(1)
2 for j = 0 to m-1 do
3     f := recycle(Ptr[j].Read())
4     B[f].Write(⊥)
5     Ptr[j].Write(f)
6 C.DWrite(0)

Operation TAS_p()
7 if (1,·) := C.DRead() then return 1
  Execute the one-time TAS()
  implementation, but replace every read
  of R_i with RRead(i), and every write of
  x to R_i with RWrite(i,x).

Operation RRead(i)
8  b := Ptr[i].Read()
9  if (·,true) = C.DRead() then return 1
10 Annc[p].Write(b)
11 if (·,true) = C.DRead() then return 1
12 ret := B[b].Read()
13 if (·,true) = C.DRead() then return 1
14 return ret

Operation RWrite(i)
15 b := Ptr[i].Read()
16 if (·,true) = C.DRead() then return 1
17 Annc[p].Write(b)
18 if (·,true) = C.DRead() then return 1
19 B[b].Write()
20 if (·,true) = C.DRead() then return 1
```

Figure 6.2: The implementation of TAS() and Reset() of the Long-Lived TAS

The implementation of our recycle(), shown in Figure 6.3, uses a shared set and two shared queues. The set data structure supports the operations add(x) which adds an element x to it, and remove() which removes and returns an arbitrary element from the set. Each shared queue supports, in addition to the standard operations enq() and deq(), an operation contains(ℓ) which returns true if element ℓ is in the queue and false otherwise. Since recycle() is called only during L.Reset() operation calls (that do not overlap), all these data structures are accessed sequentially. Moreover, since the domain of elements stored in these queues is a bounded set, a contains() operation with constant step complexity can be easily provided by using a register for each element of the domain that keeps track of the number of times the element occurs in the queue (similar to the implementation in Chapter 4). Therefore, sequential implementations with constant worst-case step complexity are used for the data structures of this operation.

Set *Free* stores indices in I_B among which a process can choose an arbitrary one to return

```
shared
Queue<int> RetQ[n] = (⊥,...,⊥)
Queue<int> AnnQ[n] = (⊥,...,⊥)
Set<int> Free = {m,···,2n + m}
Array<Boolean> Use[0...2n + m]                    // first m elements are 1, the rest 0
int Inx = 0

Operation recycle_p(int ℓ)
21 Use[ℓ] := 0
22 RetQ.enq(ℓ)
23 a := Annc[Inx]
24 AnnQ.enq(a)
25 Inx := (Inx + 1) mod n
26 y_0 := AnnQ.deq()
27 y_1 := RetQ.deq()
28 for j = 0 to 1 do
29     if y_j ≠ ⊥ then
30         if ¬Use[y_j] ∧ ¬AnnQ.contains(y_j) ∧ ¬RetQ.contains(y_j) then
31             Free.add(y_j)
32 f := Free.remove()
33 Use[f] := 1
34 return f
```

Figure 6.3: Recycle() Method of the Long-Lived `Test-And-Set`

from a `recycle()` operation. Our algorithm also guarantees that the set is never empty (see Claim 6.15 of Section 6.4). $AnnQ$ keeps track of the last n elements found in the announce array during the last n `recycle()` operations. The queue $RetQ$ stores the last n indices that are replaced by another index in an entry of Ptr. Initially, both queues contain n elements $\perp$, and the algorithm ensures that the length of each queue is n before and after each `recycle()` operation (see Claim 6.10 of Section 6.4). Finally, a boolean array $Use[0 \ldots 2n + m]$ is used to indicate for each index $\ell \in I_B$ whether the corresponding register $B[\ell]$ is in use or not.

Now consider a `recycle(`ℓ`)` call by a process p that is about to replace index ℓ in an entry of Ptr with another index. First, in line 21, the process resets the flag $Use[\ell]$ to indicate that register $B[\ell]$ will not be in use anymore, once p executes line 5 after the current `recycle()` call terminates. Then, in line 22, process p enqueues ℓ into $RetQ$. After that in lines 23–25,

p reads $Annc[Inx]$ into a local variable a, where Inx is a shared register incremented modulo n with every `recycle()` operation, and then p enqueues a to $AnnQ$. Next p dequeues an element y_0 from $AnnQ$ and an element y_1 from $RetQ$ (in lines 26–27), and thus restores the length of both queues to n. In lines 28–31, p checks whether each register $B[y_j]$ is in use (by testing the flag $Use[y_j]$) or whether an instance of y_j still appears in one of the queues, for $j \in \{0,1\}$. If any of these conditions is true, then either y_j was found announced or $B[y_j]$ was in use at some point since the last scan of the announce array begun. Thus p only adds y_j to $Free$ if none of those conditions is met. Finally, process p removes an arbitrary element f from $Free$, sets the flag $Use[f]$ to indicate that $B[f]$ will be in use when p executes line 5 after the current `recycle()` call terminates, and returns f (lines 32–34).

6.3 Long-lived Test-And-Set with Faster Reset

With the base algorithm of Section 6.2, the `Reset()` operation of the resulting long-lived TAS object takes $O(m)$ steps in the worst case, where m is the number of registers required to implement the one-time TAS object used in this transformation. Choosing the currently most space efficient one-time TAS implementation (Giakkoupis, Helmi, Higham and Woelfel, 2015), the `Reset()` takes $O(\log n)$ steps in the worst case. In this section, we propose two implementations with improved step complexity for `Reset()`: One achieves asymptotically optimal space requirement and sub logarithmic expected step complexity for both `TAS()` and `Reset()`, and another transforms any one-time TAS implementation from registers to a long-lived one, such that `Reset()` has constant step complexity in the worst case.

6.3.1 Space-Optimal and Fast Long-Lived Test-And-Set

To improve the expected step complexity of the `Reset()` implementation, we elaborately apply a *variant* of the base algorithm of Section 6.2 to a specific one-time TAS implementation by Giakkoupis and Woelfel (2012), which in turn is based on an algorithm by Alistarh and Aspnes

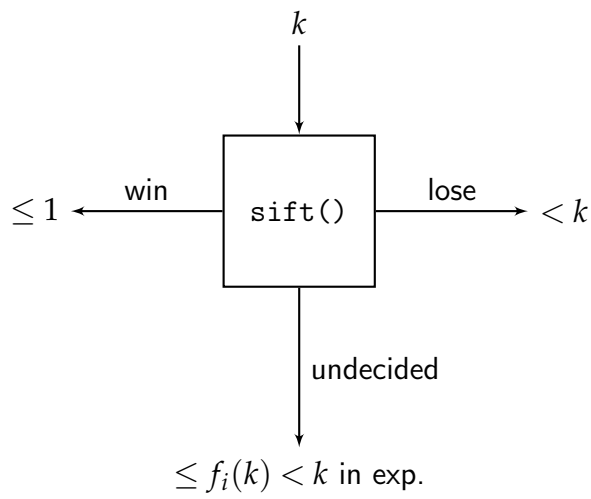

Figure 6.4: Object F_i

(2011). The shared data structure used in their implementation can be viewed as having n special randomized objects $F_1, \ldots, F_n$ and n randomized 2-process TAS objects $T_1, \ldots, T_n$. Object T_i can be implemented from constant number of registers in constant expected step complexity against the strong adaptive adversary (Tromp and Vitányi, 2002). Object F_i, as shown in Figure 6.4, supports an operation `sift()`, which each process can call at most once (before the object is reset), and which returns one of *lose*, *win*, or *undecided*. We say a process wins, loses, or is undecided at object F_i. The implementation guarantees that not all processes lose F_i, and if only one process calls F_i.`sift()`, then that process wins F_i. Moreover, the i-th object F_i is instantiated with some chosen parameter, which is a function f_i, and guarantees that if k processes call F_i.`sift()`, then in expectation at most $f_i(k) < k$ of them are undecided. Each object F_i is implemented from a constant number of registers and a `sift()` call has constant worst-case step complexity.

As shown in Figure 6.5, we can imagine that the objects $F_1, \ldots, F_n$ form a path "down" and the 2-process TAS objects $T_1, \ldots, T_n$ form a path "up". A process p walks the path down, executing `sift()` operations on each object F_i that it visits. After each `sift()` operation call on an object F_i (starting with F_1) a process decides how to proceed based on the return value: If

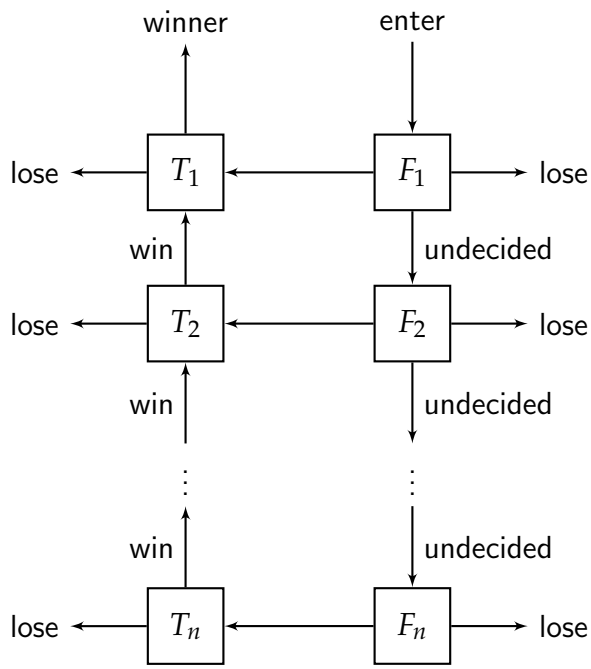

Figure 6.5: Test-and-set implementation by Giakkoupis and Woelfel (2012)

p is undecided at F_i, then it proceeds to object F_{i+1}. If it loses F_i, then it loses the entire `TAS()` operation call, and does not take any further steps. Finally, if p wins F_i, then p switches to the path "up", more precisely, it executes a `TAS()` operation call on the TAS object T_i. Whenever process p loses a `TAS()` call on some object T_i, it also immediately loses the implemented `TAS()` operation call, i.e., it returns 1. If it wins a `TAS()` call on some object T_i, then it continues to walk up the path by calling T_{i-1}.`TAS()` if $i > 1$, and if $i = 1$, then p wins the implemented `TAS()` and thus returns 0.

The functions f_i, $1 \leq i \leq n$, satisfy $f_i(k) = O(\sqrt{k})$, and therefore, in expectation only approximately $k^{1/2^i}$ processes reach object F_i. The smallest index i^*, such that no process reaches object F_{i^*} has expectation $E[i^*] = O(\log\log n)$. This yields the desired expected step complexity of $O(\log\log n)$ for any `TAS()` call (Giakkoupis and Woelfel, 2012).

Before applying the transformation of Section 6.2 to this one-time TAS implementation, we first slightly modify this implementation: We add n shared registers $L_1, \ldots, L_n$, which are initially

$\perp$. During a `TAS()` operation call, when a process wants to access F_i for the first time in that call, it first has to write a non-$\perp$ value to L_i.

Now we transform the modified object into a long-lived one using the algorithm of Section 6.2. Recall that in the resulting `Reset()` implementation, a process first writes 1 into C, and then it replaces all registers of the `TAS()` implementation whose indices are stored in Ptr with recycled ones. At the end of the `Reset()` operation call, the process writes 0 into C. Recall also that no process which executes a `TAS()` operation call can access more than one register of the `TAS()` implementation once a concurrent `Reset()` call is invoked. (This is shown in Corollary 6.18 in Section 6.4.) Hence, suppose that at the point when value 1 is written into C at the beginning of a `Reset()` call, exactly registers $L_1,\ldots,L_\ell$ have non-$\perp$ values. Then it is not possible that any object F_j, T_j, or L_{j+1}, for $j > \ell$, will be accessed by any process before C is written again at the end of that `Reset()` call.

Hence to implement `Reset()` operation, it suffices to replace the registers used in $F_1,\ldots,F_\ell$, $T_1,\ldots,T_\ell$, and $L_1,\ldots,L_{\ell+1}$ with recycled ones. In particular, during a `Reset()` call the resetter can read registers $L_1,L_2,\ldots$, until it finds the first one, $L_{\ell+1}$, with value $\perp$. Then it only replaces those registers with recycled ones that need to be replaced, which can be done in $O(\ell)$ steps, because each of $F_1,\ldots,F_n$ and $T_1,\ldots,T_n$ is implemented from a constant number of registers. As proved by Giakkoupis and Woelfel (2012), the expected value of ℓ is $O(\log\log n)$. Thus, we obtain a long-lived TAS object from $O(n)$ registers, where the step complexity of both `TAS()` and `Reset()` is $O(\log\log n)$ in expectation. This proves Theorem 6.3.

6.3.2 Long-Lived Test-And-Set with Constant Time Reset

To construct a long-lived TAS object such that a `Reset()` call requires only constant time in the worst case, one can apply the result of Theorem 4.1 from Chapter 4 to an existing one-time TAS implementation. However, the space complexity of such implementation is $O(n^2 \cdot m)$, where m is the space requirement of the original one-time TAS implementation. Here, we show how we can achieve an implementation of `Reset()` with constant step complexity, using $O(n \cdot m)$ registers.

Hoepman (1999) suggested the following straight-forward transformation of a one-time TAS object into an (inefficient) long-lived one. It is assumed that the one-time TAS object supports a `safe_reset()` operation, which resets the TAS object, provided it does not overlap with any other operation calls on that object. (This is the same as our concept of a sequentially resettable TAS object.) Processes share an array $A[0 \ldots n]$ of one-time TAS objects and one register Ptr, which stores an index $i \in \{0, \ldots, n\}$ to the one-time TAS object $A[i]$ that is in use. To execute a `TAS()`, a process reads index i from Ptr, announces that index, and reads Ptr again. If the value stored in Ptr changes between those two reads, then the process returns 1, and otherwise, it executes $A[i]$.`TAS()`. To reset the long-lived TAS object, a process reads all indices announced by other processes, and chooses an index $j \in \{0, \ldots, n\}$ that is not announced and such that the value of Ptr is not j. Then it resets the one-time TAS object $A[j]$ by executing `safe_reset()`. Finally, the process "swings the pointer" by writing j into Ptr.

The two components that make this `Reset()` call inefficient are reading the announce array during each `Reset()` call, and sequentially resetting the one-time TAS object. We augment techniques introduced earlier in this work (in Section 6.2.2 and Section 4.5) to this simple algorithm to achieve a `Reset()` implementation with optimal step complexity of $O(1)$. In particular, the first idea is to use the recycling technique proposed in Section 6.2.2 to choose the index of the next one-time TAS to put into use during a `Reset()` call in constant number of steps, instead of reading the entire announce array in $O(n)$ steps as suggested by Hoepman (1999). This change requires a $\Theta(n)$ additional one-time TAS objects. Our second idea is to use the result of the implementation of Section 4.5. With this idea, given any one-time TAS object implemented from registers, we can obtain a sequentially resettable TAS with one additional register, such that the sequential reset, that is `safe_reset()`, takes $O(1)$ steps in the worst case (see Corollary 4.3).

Augmenting these modules to the simple "swinging the pointer" technique allow a long-lived TAS implementation from any one-time TAS implemented from registers. The resulting implementation uses $O(n)$ instances of the one-time TAS object. The `TAS()` operation on the

long-lived object requires a constant number of additional steps, and `Reset()` has $O(1)$ step complexity in the worst case. This shows how Theorem 6.4 is obtained.

6.4 Correctness of the Base Algorithm

Consider the long-lived TAS implementation L of Figures 6.2 and 6.3 obtained from a one-time TAS implementation. We say a transcript is *permissible on* L if it can arise from a sequence of L.`TAS()` and L.`Reset()` operation calls, where L.`Reset()` can be invoked only by a process that has won an L.`TAS()` call and not subsequently called L.`Reset()`.

Our main lemma proves that the interpreted history of any permissible transcript on L, in which no two `Reset()` calls overlap, has a linearization, with some specific properties:

Lemma 6.6. *For any permissible transcript Λ^* on L, in which no two* `Reset()` *calls overlap, history $H = \Gamma(\Lambda^*)$ has a linearization S, satisfying:*

(a) all complete `Reset()` *calls in H linearize at their responses, and*

(b) if H contains a pending `Reset()`*, then it is the last operation call to linearize.*

The lengthy proof of this lemma is deferred to Section 6.4.1. Assuming that this lemma holds, we now show in Lemma 6.7 that no two `Reset()` calls can overlap in any permissible transcript on L. Therefore, Lemmas 6.6–6.7 together prove linearizability of any permissible transcript obtained from the long-lived TAS implementation L of Figures 6.2 and 6.3.

Lemma 6.7. *In any permissible transcript on L, no two* `Reset()` *calls overlap.*

Proof. Suppose for the sake of contradiction that there are overlapping `Reset()` calls in a permissible transcript Λ on L. Let E_1 by p_1 and E_2 by p_2 be some `Reset()` calls that overlap each other, where $p_1 \neq p_2$. Suppose E_1 is invoked before E_2, and no two `Reset()` calls overlap before the invocation of E_2. Let Λ' be the longest prefix of Λ in which no two `Reset()` calls

overlap. This implies that Λ' ends just before the invocation of E_2. Then E_1 is pending in transcript Λ'. By Lemma 6.6, Λ' has a linearization S, such that

$$E_1 \text{ is the last operation call in } S. \tag{6.1}$$

Since Λ is a permissible transcript, and process p_2 invokes the `Reset()` operation E_2 in Λ immediately after the prefix Λ' of Λ completes, p_2's last operation call in Λ' is a `TAS()` call M_2 that returns 0 before Λ' ends. Then p_2's last operation call in S is also M_2. Since S is a valid sequential history on a TAS object, if there is a `Reset()` call after M_2 in S, then the first such `Reset()` call must be also by p_2. However, M_2 is p_2's last operation call in S, and by (6.1) `Reset()` call E_1 by process p_1 appears after M_2 in S. This is a contradiction since we showed $p_1 \neq p_2$. □

Lemmas 6.6–6.7 imply that any permissible history on L is linearizable.

Corollary 6.8. *Any permissible history on L is linearizable.*

6.4.1 Proof of Lemma 6.6

Assume Λ^* is a permissible transcript on L in which no two `Reset()` calls overlap. We assume w.l.o.g. (for the purpose of proving linearizability) that Λ^* is finite: Suppose Λ^* is infinite but $\Gamma(\Lambda^*)$ is not linearizable. Then there must be a finite prefix of Λ^*, such that the interpreted history of that prefix is not linearizable. Therefore, it suffices to prove for every finite prefix that its interpreted history is linearizable.

Also assume that Λ^* contains $k \geq 0$ `Reset()` operation calls $E_1, \ldots, E_k$, such that $t_{E_i@inv} < t_{E_{i+1}@inv}$, for $i \in \{1, \ldots, k-1\}$. Moreover, assume all except possibly E_k complete in Λ^*. If E_k is pending at the end of Λ^*, we let E_k run to completion (which is possible because `Reset()` is wait-free), and Λ denote the resulting transcript, and otherwise we let $\Lambda = \Lambda^*$. In the rest of this section, we prove $\Gamma(\Lambda)$ has a linearization, and so $\Gamma(\Lambda^*)$, which is a prefix of $\Gamma(\Lambda)$, also has a linearization.

Recall that a C.`DWrite(1)` call and a C.`DWrite(0)` call are executed as the first step following the invocation and as the last step preceding the response of each `Reset()` call E_i, respectively. Let t_0 be the starting point of Λ. Also for $i \in \{1,\ldots,k\}$, let t_{2i-1} denote the point at which the C.`DWrite(1)` call of E_i is executed, and t_{2i} be the point at which the C.`DWrite(0)` of E_i is executed. (For the purpose of the linearizability proofs, we assume C is atomic. This assumption can be made without loss of generality because replacing atomic base objects with linearizable ones preserves linearizability of the implemented objects.) Therefore, $t_{2i-1} = t_{E_i@inv}$ and $t_{2i} = t_{E_i@rsp}$, for $i \in \{1,\ldots,k\}$. Finally let t_{2k+1} be an arbitrary point after all operation calls in Λ have ended, or $t_{2k+1} = \infty$ if Λ is not complete. Since no two `Reset()` calls overlap in Λ, we have $t_{E_i@rsp} < t_{E_{i+1}@inv}$, for all $i \in \{1,\ldots,k-1\}$, and so

$$t_0 < t_1 < \cdots < t_{2k} < t_{2k+1}. \tag{6.2}$$

Therefore, for $i \in \{1,\ldots,k\}$, and $j \in \{0,\ldots,2k\}$, we have

(a) $C = 0$ at t_0,

(b) value 1 gets written to C at t_{2i-1},

(c) value 0 gets written to C at t_{2i}, and

(d) no process writes to C during (t_j, t_{j+1}).

Thus, we can conclude the following.

Observation 6.9. *Let t be any point in time. Then $C = 1$ at t if and only if there is an index $i \in \{1,\ldots,k\}$, such that $t \in [t_{2i-1}, t_{2i})$.*

In order to prove Lemma 6.6, we need to prove several claims. First, we show that at any point in time when no process is executing lines 22–27 of a `recycle()` call, each of $RetQ$ and $AnnQ$ stores n elements.

Claim 6.10. *Let t be a point at which either a process executes a* `recycle()` *operation call E and $t \notin [t_{E@24}, t_{E@27}]$, or no process executes a* `recycle()` *operation call. Then each of the queues $RetQ$ and $AnnQ$ has size exactly n.*

Proof. Each of these queues initially has n elements. During lines 22–27 of each `recycle()` operation call, one element is enqueued to each of $RetQ$ and $AnnQ$ in lines 22 and 24, respectively, and one element is dequeued from each of those queues in lines 26 and 27. As no two `recycle()` operation calls overlap, the size of each queue is exactly n, when a process finishes line 27. Moreover, these queues are not modified anywhere else, hence, the size of queues $RetQ$ and $AnnQ$ are exactly n whenever no `recycle()` operation call is pending in lines 22–27. □

Next, we examine what happens once an index is enqueued to one of $RetQ$ or $AnnQ$.

Claim 6.11. *Suppose a process enqueues an index $b \in I_B$ into one of the queues $RetQ$ or $AnnQ$ at some point t during a* `recycle()` *operation call R. Let t' be the point right after $n-1$* `recycle()` *calls following R completed, or $t' = \infty$ if this does not happen in Λ. Then no process adds b to $Free$ throughout $[t, t')$.*

Proof. By Claim 6.10, the size of each of the queues $RetQ$ and $AnnQ$ is n just before t. Both $RetQ$ or $AnnQ$ are only modified in a `recycle()` call. During a `recycle()` operation call, a process dequeues one element from each of these queues (lines 26–27). Therefore, if a copy of index b is enqueued into a queue at time t during R, then this copy remains in the same queue during R and the following at least $n-1$ `recycle()` operation calls, and so throughout $[t, t')$, because no two `recycle()` calls overlap. An index b can only get added to $Free$ in line 31 of a `recycle()` call if no instance of b is in any of these queues and $Use[b] = 0$. Hence, any time line 29 gets executed for $y_j = b$ during $[t, t')$, the if-condition evaluates to false and so index b does not get added to $Free$. □

Using Claim 6.11, we now show that if a process reads an index from $Annc[p]$, for some p, then that index is not added to $Free$ until the value of $Annc[p]$ changes.

Claim 6.12. *Suppose some process reads $b \in I_B$ from $Annc[p]$ at some point t_1 during a* `recycle()` *operation call, for some p. Let t_2 be the first point after t_1 at which $Annc[p] \neq b$, or $t_2 = \infty$ if such a point does not exist. Then, no process adds b to $Free$ throughout $[t_1, t_2)$.*

Proof. Assume for the sake of contradiction that at some point $t \in [t_1, t_2]$, some process adds b to $Free$ in line 31 of some `recycle()` call R. Let $t' \in [t_1, t_2]$ be the last point before t at which some process reads b from $Annc[p]$ during some `Reset()` call R'. Since a process reads b from $Annc[p]$ at t_1, such a point t' exists in this interval. Therefore at t', a process executes line 23 of R', and it then enqueues b into $AnnQ$ in line 24 (recall that no two `Reset()` calls overlap). Thus by Claim 6.11, index b can only be added to $Free$ after at least $n-1$ `recycle()` calls that follow R' complete. This implies that $n-1$ `recycle()` calls complete after R' and before R, since we assumed some process adds b to $Free$ in line 31 of R. Hence, lines 23–25 of the `recycle()` operation get executed at least n times after R' and before line 31 of R and so during (t', t). Since in each `recycle()` call, Inx gets incremented modulo n in line 25 and does not get modified anywhere else, line 23 of `recycle()` gets executed at least once when $Inx = p$ during (t', t). Therefore, index b is read from $Annc[p]$ in that time, because $Annc[p] = b$ throughout $(t', t) \subseteq [t_1, t_2]$. This contradicts the assumption that this happens at point t' for the last time before t. □

Next in Claim 6.13, we state some invariants about the relation between Ptr, Use, and $Free$. Using Claim 6.13, we show, in Claim 6.14, that there are at most m indices $b \in I_B$, such that $Use[b] = 1$, just before a `recycle()` call is invoked. For both Claim 6.13 and Claim 6.13, we assume that $Free$ is never empty when a process is poised to remove an index from this set in line 32. Later in the proof of Claim 6.15, we show that this is in fact true.

Claim 6.13. *Consider some indices $b \in I_B$ and $j \in \{0, \ldots, m-1\}$. Suppose $Free \neq \emptyset$ when a process is poised to execute line 32. Then at any point, if $Ptr[j] = b$, then all of the following are true.*

(a) $Ptr[j'] \neq b$, *for any* $j' \in \{0,\ldots,m-1\} \setminus \{j\}$,

(b) either $Use[b] = 1$ *or a process is poised to execute one of lines 22–33 or 4-5, and*

(c) $b \notin Free$.

Proof. Initially, $Free = \{m,\ldots,2n+m\}$, and for any $j \in \{0,\ldots,m-1\}$, $Ptr[j] = j$ and $Use[j] = 1$. Thus, the claim holds at t_0.

Now consider some point $t > t_0$, such that at least one step is executed in $[t_0,t)$. Let p be the process taking the last step prior to t (such a step exists because $t \neq t_0$). Moreover, assume

$$\text{Claim 6.13 is true throughout } [t_0,t). \tag{6.3}$$

We show that Claim 6.13 is also true at t.

Case 1. Suppose that at t, for some $j_1 \in \{0,\ldots,m-1\}$, process p writes b into $Ptr[j_1]$ in line 5 of a `Reset()` call E, and one of properties (a)-(c) is untrue. Also let R be the last `recycle()` call p executes in line 3 prior to t. Since p writes index b into $Ptr[j_1]$ at t,

$$\texttt{recycle()} \text{ call } R \text{ returns } b. \tag{6.4}$$

First we show that (a) holds at t. Suppose not, then there is an index $j_2 \in \{0,\ldots,m-1\} \setminus \{j_1\}$, such that $Ptr[j_2] = b$ at t. No `Reset()` call overlaps E, and process p does not write to $Ptr[j_2]$ during $[t_{R@inv},t]$, therefore,

$$Ptr[j_2] = b \text{ throughout } [t_{R@inv},t]. \tag{6.5}$$

By (6.4), R returns b, and so p removes b from $Free$ at $t_{R@32}$. Thus, $b \in Free$ right before $t_{R@32}$. However, by (6.3) and (6.5), property (c) holds just before $t_{R@32}$, and so, $b \notin Free$ at that point, which is a contradiction.

Now we prove that (b) and (c) hold at t. By (6.4), R returns b, and so p removes b from $Free$ in line 32, and writes 1 into $Use[b]$ in line 33 of R. No `Reset()` call overlaps E, and process p does not add any index to $Free$ during $[t_{R@32},t]$, and it does not change the value of $Use[b]$ during $[t_{R@33},t]$. Therefore, $Use[b] = 1$ and $b \notin Free$ at t, and so both (b) and (c) hold at t.

Case 2. Next, suppose that p's last step before t is a step other than writing b into an entry of Ptr. Suppose

$$Ptr[j] = b \text{ at } t. \tag{6.6}$$

Since property (a) is true before t, and Ptr remains unchanged, (a) is also true at t. Hence, suppose that one of properties (b) and (c) is untrue at t. If p's step is a write of 0 into $Use[b]$, then p executes line 21 just before t, and so at t, process p is poised to execute line 22. Thus, property (b) holds. Hence, in its step right before t, p either executes line 5 (and writes $b' \neq b$ to $Ptr[j^*]$, $j^* \in \{0,\ldots,m-1\} \setminus \{j\}$), or it adds b to $Free$.

Case 2.1. Consider the case that at t, process p finishes executing line 5 of some iteration j^* of the loop of some `Reset()` call E, and property (b) does not hold. Thus,

$$Use[b] = 0 \text{ at } t. \tag{6.7}$$

Let t' be the last point at which p is poised to execute line 3 prior to t. No `Reset()` call overlaps E, so no process other than p can write to an entry of Ptr during $[t',t]$. Since p's last step prior to t is not a write of b into an entry of Ptr, p writes some index $b' \neq b$ to $Ptr[j^*]$ at t. Process p does not write to any other entry of Ptr during $[t',t]$. Thus, by (6.6),

$$Ptr[j] = b \text{ throughout } [t',t] \text{ and } j \neq j^*. \tag{6.8}$$

By (6.3) and (6.8), properties (a) and (b) imply that

$$Ptr[j^*] \neq b \text{ and } Use[b] = 1 \text{ at } t'. \tag{6.9}$$

According to (6.7), $Use[b] = 0$ at t, hence by (6.9) and since no `Reset()` call overlaps E, p writes 0 to $Use[b]$ during $[t',t]$, and it can do so only in line 21 of R. This implies that R is a `recycle(`b`)` call. Thus by line 3, $Ptr[j^*] = b$ at t'. This contradicts (6.9).

Case 2.2. Suppose that in its step right before t, p adds b to $Free$ in line 31 of a `recycle()` call R. Thus, $Use[b] = 0$ when p reads this value in line 30 of R prior to t. Let t' be the point

at which p is poised to execute line 21 of R. By (6.6), $Ptr[j] = b$ at t. Moreover, p does not write to any entry of Ptr during $[t', t]$, and no `Reset()` overlaps the current `Reset()` call of p during which R is executed. So $Ptr[j] = b$ at t' as well. Therefore, by (6.3), property (b) implies that $Use[b] = 1$ at t'. Hence, p writes 0 to $Use[b]$ at some point during $[t', t]$, and p can only do so in line 21 of R. Thus, in line 22 of R, p enqueues a copy of b to $RetQ$. By Claim 6.11, no process adds b to $Free$ during R. This contradicts the assumption that p adds b to $Free$ in line 31 of R. □

Claim 6.14. *Suppose $Free \neq \emptyset$ when a process is poised to execute line 32. Then just before the invocation of a* `recycle()` *call, there are at most m indices $b \in I_B$, such that $Use[b] = 1$.*

Proof. Initially, there are m indices $b \in I_B$, such that $Use[b] = 1$. Since Use is only modified during a `recycle()` call, the claim holds just before the invocation of the first `recycle()` call in Λ. During each `recycle(`b`)` call, $Use[b]$ is set to 0 in line 21, and one entry of Use is set to 1 in line 33. Thus, it is enough to show that for any $b \in I_B$, $Use[b] = 1$ just before a process writes 0 into $Use[b]$ in line 21 of a `recycle(`b`)` call.

Consider a `recycle(`b`)` call R by some process p, for some $b \in I_B$. By line 3, just before p executes line 21 of R, $Ptr[j] = b$, for some $j \in \{0, \ldots, m-1\}$. Thus by Claim 6.13(b), $Use[b] = 1$ at that point. □

Now, we prove that $Free$ contains at least one index, when a process is poised to remove an element from this set.

Claim 6.15. *At any point, for any process p, and any index $b \in I_B$, the following holds.*

(a) If p is poised to execute line 32, then there is at least one index in $Free$, and

(b) either p is poised to execute one of lines 22–31 or line 33, or $Use[b] = 1$, or $b \in RetQ \cup AnnQ \cup Free$.

Proof. Initially, for any index $b \in \{0, \ldots, m-1\}$, we have $Use[b] = 1$, and for any index $b \in I_B \setminus \{0, \ldots, m-1\}$, we have $b \in Free$. So the claim holds at t_0.

Suppose for some point $t > t_0$,

$$\text{properties (a) and (b) are true throughout } [t_0, t). \tag{6.10}$$

Case 1. First we prove that at t, property (a) is still true. Suppose at t, some process p becomes poised to execute line 32 of some `recycle()` call R, while $Free$ is empty. Let t' be the point at which p becomes poised to execute line 21 of R for the last time before t. By (6.10), property (b) holds at t'. Thus, for each index $b \in I_B$, either $Use[b] = 1$, or $b \in RetQ \cup AnnQ \cup Free$ at t'. By Claim 6.10, each of $RetQ$ and $AnnQ$ contains n elements at t'. According to (6.10), property (a) holds at any point before t. Therefore, by Claim 6.14, there are at most m indices $b \in I_B$, such that $Use[b] = 1$ at at any point before t, and so at t'. Hence, since $|I_B| = 2n + m + 1$, there is at least one index of I_B that is in $Free$ at that point. Since only in line 32 an element can be removed from $Free$, and no process executes that line throughout $[t', t)$, no element gets removed from $Free$ during that interval. Thus, there is at least one index in $Free$ at t.

Case 2. Next we prove that at t, property (b) holds. Suppose not, then no process is poised to execute any of lines 22–31 or line 33, and there is an index $b \in I_B$, such that

$$Use[b] = 0 \text{ and } b \notin RetQ \cup AnnQ \cup Free, \text{ at } t. \tag{6.11}$$

Property (b) cannot get invalidated right after modifying $RetQ$ and $AnnQ$, or writing 0 to $Use[b]$, or removing an element from $Free$, because a process becomes poised to execute one of lines lines 22–31 and line 33. Thus, there is some process p, such that at t, p finishes executing either line 31, or line 33.

Case 2.1. Suppose p finishes executing line 31 of some `recycle()` call R at t. Let $t' < t$ be the point at which p is poised to execute line 21 of R. By (6.10), property (b) holds at t', and so at that point either $Use[b] = 1$ or $b \in RetQ \cup AnnQ \cup Free$.

First assume $Use[b] = 0$ at t'. Thus, $b \in RetQ \cup AnnQ \cup Free$. By (6.11) and since no `recycle()` call overlaps R, process p must dequeue the last instance of b from one of the

queues or removes b from $Free$ during $[t',t)$. Process p does not remove any index from $Free$ during this interval, thus p dequeues the last instance of b in the queues from one of them in line 26 or line 27 of R. $Use[b] = 0$ at t', and it does not change before t, thus the if-condition in line 30 of R evaluates to true for b, and so p adds b to $Free$ in line 31 before t. This index remains in $Free$ until at least t, which contradicts (6.11).

If $Use[b] = 1$ at t', then by (6.11) and since no `recycle()` call overlaps R, process p writes 0 into $Use[b]$ at some point during $[t',t]$ and it can do so only in line 21 of R. Then in line 22 of R, p enqueues a copy of b into $RetQ$. By Claim 6.10, this instance of b remains in the queue, until line 27 has been executed n times after that. Since no `recycle()` call overlaps R, line 27 is executed only once during $[t_{R@22},t]$. Thus, an instance of b is in $RetQ$ at t, which contradicts (6.11).

Case 2.2. Suppose p finishes executing line 33 of a `recycle()` call R at t. Let t' be a point when p is poised to execute line 32 of R. By (6.10), properties (a) and (b) hold at t'. Thus, there is at least one index in $Free$ at t', and either $Use[b] = 1$ or $b \in RetQ \cup AnnQ \cup Free$. The queues $RetQ$ and $AnnQ$ are not affected by p's steps in lines 32–34, and thus do not change in $[t',t]$. Moreover, if the index that p removes from $Free$ in line 32 of R is not b, then $Use[b]$ does not change. Thus, either $Use[b]$ remains equal to 1, or b remains in $RetQ \cup AnnQ \cup Free$ at t. This contradicts (6.11). If p removes b from $Free$ in line 32 of R, then it also sets $Use[b]$ to 1 in line 33, and so $Use[b] = 1$ at t, which contradicts (6.11). □

We now discuss what happens after an index some process writes an index b into an entry of Ptr. Recall that by Claim 6.13, if $Ptr[i] = b$, then b is not in $Free$.

Claim 6.16. *Suppose at some point t, no* `Reset()` *call is pending and $Ptr[j] = b \in I_B$, for some $j \in \{0,\ldots,m-1\}$. If there is a point t' at which $b \in Free$ for the first time after t, then*

(a) a `Reset()` *call is invoked during (t,t'), and*

(b) at least n `recycle()` calls complete during $[t_{E'@inv}, t']$, where E' is the first `Reset()` call that is invoked after t.

Proof. By Claim 6.15(a) and Claim 6.13(c), we have

$$b \notin Free \text{ at } t, \tag{6.12}$$

but it is assumed that $b \in Free$ at t'. Thus, some process adds b to $Free$ during a `Reset()` call at some point during (t, t'). This `Reset()` call is invoked during (t, t'), because no `Reset()` call is pending at t. Therefore Part (a) follows.

According to (6.12), $b \notin Free$ at t, and by the claim assumption b is in $Free$ at t' for the first time after t. This implies that at t, some process p adds b to $Free$ in line 31 of a `recycle()` call R. Thus, $Use[b] = 0$ when p executes the if-condition in line 30 for the last time before t. By Claim 6.15(a) and Claim 6.13(b), and since no `Reset()` call is pending at t, $Use[b] = 1$ at that point. Thus, some process writes 0 to $Use[b]$ in line 21 of a `recycle(`b`)` call R' at some point during (t, t'). In line 22 of R', index b is enqueued to $RetQ$ and therefore, by Claim 6.11, index b can only be added to $Free$ after at least $n-1$ `recycle()` calls following R' complete. Therefore, at least n `recycle()` calls complete during $[t_{R'@inv}, t']$. Operation call R' is invoked after t and in a `Reset()` call. Since no `Reset()` call is pending at t, and by the assumption, E' is the first `Reset()` call invoked after t, we have $t_{R'@inv} > t_{E'@inv}$. Thus, at least n `recycle()` calls complete during $[t_{E'@inv}, t']$, and so Part (b) also holds. □

The following claim shows that if a process executes a C.`DRead()` call during a `TAS()` operation call while a `Reset()` is pending, then the `TAS()` call immediately returns 1 before executing any further operations on a shared variable.

Claim 6.17. *Consider a `TAS()` operation call M by some process p, and let $i \in \{1, \ldots, k\}$. Suppose at some point $t \in [t_{2i-1}, t_{2i})$, process p executes a C.`DRead()` call during M. Then immediately after the `DRead()`, M returns with value 1.*

Proof. Process p can execute C.DRead() in line 7 of M or during an RRead() or an RWrite() call executed in M in one of lines 9, 11, 13, 16, 18 and 20. First suppose process p executes line 7 of M at t. Then by Observation 6.9, p reads $(1,\cdot)$ at t, and so the if-condition in line 7 evaluates to true and p immediately returns value 1.

Now suppose p executes C.DRead() in one of lines 9, 11, 13, 16, 18 and 20 of an RRead() or an RWrite() call executed in M at $t \in [t_{2i-1}, t_{2i})$. Let t' be the last time p executes a C.DRead() call before t_{2i-1} during M. Such a point t' exists, because p must have executed the C.DRead() call in line 7 of M before t_{2i-1}, since otherwise, as we just showed, M would have returned 1 before t. Hence $t_{2i-1} \in (t', t)$. By the definition of t_{2i-1}, some process executes a C.DWrite(0) call at $t_{2i-1} = t_{E_i@inv} = t_{E_i@1}$. Therefore, the C.DRead() call executed at t must return $(\cdot, \texttt{true})$, indicating a C.DWrite() has happened since the preceding C.DRead() call at t'. Thus, the if-condition following the DRead() of C at t evaluates to true, and p immediately returns 1. □

Corollary 6.18. *After a* Reset() *call is invoked by some process during a* TAS() *call by process p, p executes at most one operation on B before its* TAS() *completes.*

Proof. Consider some TAS() call M by some process p, and suppose the i-th Reset() call E_i in Λ is invoked at $t_{E_i@inv} = t_{2i-1}$ during the execution of M, for some $i \in \{0, \ldots, m-1\}$. As p executes C.DRead() after and before accessing an entry of B, and by Claim 6.17, p can access at most one register of B after the invocation of E_i and before p reads C again when it detects the overlapping Reset() call, and returns 1. □

Using Claims 6.12 and 6.16, the following claim prove an important property of our algorithm: If a process is about to access a register $B[b]$ during a TAS() call, then $B[b]$ has not been reset since the invocation of the current TAS() call. Equivalently, this claim shows that if a process is about to reset a register $B[b]$, then this register is not going to be accessed until it is in use again. The idea is that if a process p is about to access $B[b]$, then no Reset() call is invoked before

the point at which p announces b, because otherwise p would detect the overlapping `Reset()` call when it reads C after announcing b. Moreover, if an element of Ptr has value b when some process p announces b, then the next n `recycle()` calls will not add b to $Free$ and during those n `recycle()` calls some process reads p's announcement of index b, and so register $B[b]$ does not get reset until p changes its announcement.

Claim 6.19. *Consider a* `TAS()` *operation call M by p, and some index $b \in I_B$. Suppose process p executes $B[b]$.*`Read()` *or $B[b]$.*`Write()` *at some point t during M. Then no process writes $\perp$ to $B[b]$ in line 4 of a* `Reset()` *call during interval $[t_{M@inv}, t]$.*

Proof. Suppose p executes $B[b]$.`Read()` or $B[b]$.`Write()` during an `RRead()`, respectively, an `RWrite()` operation call Op in M. Thus, p must have read b from $Ptr[i]$, for some $i \in \{0, \ldots, m-1\}$, at an earlier point during Op (in line 8, respectively, line 15 of Op).

Let t_a be the point at which p writes b into $Annc[p]$ during Op, that is $t_a = t_{Op@10}$ if Op is an `RRead()`, and $t_a = t_{Op@17}$ if Op is an `RWrite()` operation call. Then p does not overwrite this value until some time after t. Therefore,

$$Annc[p] = b \text{ throughout } [t_a, t]. \tag{6.13}$$

Process p accesses $B[b]$ at t, thus neither of the if-conditions of Op executed before t (in lines 9 and 11 if Op is an `RRead()`, and in lines 16 and 18 if Op is an `RWrite()`) evaluates to true. Moreover, C must have value 0 at $t_{M@7}$. Therefore, $C = 0$ throughout $[t_{M@7}, t_a] = [t_{M@inv}, t_a]$. Hence, no `Reset()` is pending at the invocation of M and no `Reset()` gets invoked after the invocation of M and before t_a, i.e.

$$\text{no } \texttt{Reset()} \text{ call overlaps } [t_{M@inv}, t_a]. \tag{6.14}$$

Thus, no process writes $\perp$ to $B[b]$ in line 4 of a `Reset()` call during $[t_{M@inv}, t_a]$. In the rest of this proof we show that

$$\text{no process writes } \perp \text{ to } B[b] \text{ in line 4 of a } \texttt{Reset()} \text{ call during } (t_a, t], \tag{6.15}$$

which completes the proof of this claim.

To prove (6.15), we show that

$$b \notin \mathit{Free} \text{ throughout } (t_a, t]. \tag{6.16}$$

To see why this is enough, suppose that (6.16) is true, but (6.15) is not. Then a process writes $\perp$ to $B[b]$ in line 4 of a `Reset()` call E during $(t_a, t]$. Hence, a `recycle()` call R executed during E must return b. By (6.14), E is invoked after t_a, and so R is also invoked after t_a. However by (6.16), when line 32 of R is executed, $b \notin \mathit{Free}$, and so R does not return b. This is a contradiction.

Proof of (6.16). By (6.14), no process changes the value stored in $\mathit{Ptr}[i]$ between the point at which p reads Ptr during Op and t_a. Thus,

$$\mathit{Ptr}[i] = b \text{ at } t_a. \tag{6.17}$$

Therefore, by Claim 6.15(a) and Claim 6.13(c), $b \notin \mathit{Free}$ at t_a. Next, we show that no process adds b to Free during $(t_a, t]$. Suppose for the sake of contradiction that at some point $t^* \in (t_a, t]$ some process adds b to Free for the first time after t_a. Then by (6.17), (6.14), and Claim 6.16(a) a `Reset()` call is invoked between (t_a, t^*). Let E' be the first `Reset()` call executed during this interval. By (6.14),

$$t_a < t_{E'@inv} < t^* < t. \tag{6.18}$$

Moreover, by Claim 6.16(b), at least n `recycle()` calls complete during $[t_{E'@inv}, t^*]$.

Now consider the first n of these at least n `recycle()` calls. By line 25, Inx gets incremented modulo n during each `recycle()` call. Therefore, there is a `recycle()` call R_r among these n `recycle()` calls, such that $\mathit{Inx} = p$ at the invocation of R_r. By (6.13) and (6.18), $\mathit{Annc}[p] = b$ throughout $[t_{E'@inv}, t^*] \subseteq [t_a, t]$. Therefore, a process reads b from $\mathit{Annc}[p]$ during R_r at some point $t_r \in [t_{E'@inv}, t^*]$. By Claim 6.12, index b does not get added to Free throughout $[t_r, t]$, because by (6.13) $\mathit{Annc}[p]$ does not change in that interval. Thus no process adds b to Free at $t^* \in [t_r, t]$. This is a contradiction. $\square$

Recall the definition of points $t_0, t_1, \ldots, t_{2k+1}$ discussed on page 130. For any $i \in \{0, \ldots, k\}$, let I_i denote time interval $[t_{2i}, t_{2i+1})$, and let $\mathcal{O}_i$ be the set of all `TAS()` operation calls M in Λ, such that $t_{M@inv} \in I_i$. Moreover, for $i \in \{1, \ldots, k\}$, let $\mathcal{F}_i$ be the set of all `TAS()` operation calls M in Λ, such that $t_{M@inv} \in [t_{2i-1}, t_{2i})$. We let $\mathcal{B}_i$ denote the set of all registers that are in use (pointed by Ptr) during time interval I_i, for $i \in \{0, \ldots, k\}$. Since the values stored in array Ptr only change during interval (t_{2i-1}, t_{2i}) of a `Reset()` operation call E_i, for $i \in \{1, \ldots, k\}$, $\mathcal{B}_i$ is the set of all registers whose indices are written to an entry of Ptr during `Reset()` call E_i, and $\mathcal{B}_0$ is the set of all registers that are in use initially.

In the following three claims, we establish some properties regarding these sets. In particular, Claim 6.20 states that any `TAS()` call M returns 1 provided that $M \in \mathcal{F}_i$, or M is in $\mathcal{O}_i$ and responds after E_{i+1} is invoked (i.e. after t_{2i+1}). Next in Claim 6.21, we show that during I_i only operations in $\mathcal{O}_i$ access registers of $\mathcal{B}_i$, and these operations do not access any other registers of B. Claim 6.22 says that right after E_i responds at t_{2i}, all registers of $\mathcal{B}_i$ are in their initial states.

Claim 6.20. *Consider some* `TAS()` *operation call M in Λ,*

(a) if $M \in \mathcal{F}_i$, for some $i \in \{1, \ldots, k\}$, then M returns 1, without executing any operation call on any register of B, and

(b) if $M \in \mathcal{O}_i$, for some $i \in \{0, \ldots, k\}$, and $t_{M@rsp} > t_{2i+1}$, then M returns 1.

Proof. Let p be the process that executes operation call M. First suppose $M \in \mathcal{F}_i$. Thus, we have $t_{M@inv} \in [t_{2i-1}, t_{2i})$. Process p executes a C.`DRead()` call at $t_{M@inv}$ (in line 7). Therefore, by Claim 6.17, p returns 1 at $t_{M@inv}$. Hence, p executes no operation on any register of B, which proves Part (a).

Now suppose $M \in \mathcal{O}_i$, and $t_{M@rsp} > t_{2i+1}$. That is $t_{M@inv} < t_{2i+1} < t_{M@rsp}$. By the implementation, p executes C.`DRead()` at $t_{M@inv}$, and during the remainder of M after each operation call on a shared object other than C. Hence, at least one C.`DRead()` call is executed before t_{2i+1} and at least one C.`DRead()` call is executed after t_{2i+1} during M. Since a C.`DWrite()` is ex-

ecuted at t_{2i+1}, the first C.DRead() that is executed after t_{2i+1} returns $(\cdot,\texttt{true})$, and so M immediately returns 1. $\square$

Claim 6.21. *Let p be a process and M an operation call by p in which p accesses a register $B[j]$, for some $j \in I_B$, at point $t \in I_i$, for $i \in \{0,\ldots,k\}$. Then $B[j] \in \mathcal{B}_i$ if and only if $M \in \mathcal{O}_i$.*

Proof. We first show that

$$M \text{ is a } \texttt{TAS()} \text{ call.} \tag{6.19}$$

No Reset() call is invoked before t_1, hence no process accesses any register of B in a Reset() call during $I_0 = [t_0, t_1)$. Now assume that $i \in \{1,\ldots,k\}$. By definition, $I_i = [t_{2i}, t_{2i+1})$ and so if $i < k$, then $I_i = [t_{E_i@rsp}, t_{E_{i+1}@inv})$, and if $i = k$, then $I_i = [t_{E_k@rsp}, t_{2k+1})$. In both cases, the only Reset() call in which a process takes any shared memory step during $I_i = [t_{2i}, t_{2i+1})$ is E_i, and that is only at $t_{2i} = t_{E_i@rsp}$. However, the last shared memory step of a Reset() call is not accessing any register of B. Therefore, no process accesses any register of B in a Reset() call during I_i.

Suppose $B[j] \in \mathcal{B}_i$, then we prove that $M \in \mathcal{O}_i$. According to (6.19), M is a TAS() call. Now suppose for the sake of contradiction that $M \notin \mathcal{O}_i$. Therefore, $t_{M@inv} \notin [t_{2i}, t_{2i+1})$. Since p accesses $B[j]$ in M during interval $[t_{2i}, t_{2i+1})$, thus $t_{M@inv} < t_{2i}$, and $i > 0$.

First suppose that $t_{M@inv} \in [t_{2i-1}, t_{2i})$, and therefore, by definition, we have $M \in \mathcal{F}_i$. Then, by Claim 6.20(a), M returns 1 before p executes any operations on any register of B, which is a contradiction. Now, suppose that $t_{M@inv} < t_{2i-1}$. By Claim 6.19, no process writes $\perp$ into $B[j]$ in line 4 of a Reset() call during $[t_{M@inv}, t]$, because p accesses $B[j]$ during M at point t. Since $t_{M@inv} < t_{2i-1}$ and $t_{2i} < t$, we have $[t_{2i-1}, t_{2i}] \subseteq [t_{M@inv}, t]$. Thus,

$$\text{no process writes } \perp \text{ into } B[j] \text{ in line 4 of a } \texttt{Reset()} \text{ call during } [t_{2i-1}, t_{2i}]. \tag{6.20}$$

Since $B[j] \in \mathcal{B}_i$, by definition, index j is written to an entry of Ptr in Reset() call E_i. However, before that, in line 5 of E_i, register $B[j]$ is reset in line 4 at some point during $(t_{E_i@inv}, t_{E_i@rsp}) = (t_{2i-1}, t_{2i})$. This contradicts (6.20).

Next suppose that $M \in \mathcal{O}_i$, then we prove that $B[j] \in \mathcal{B}_i$. Suppose not. Process p must read index j from Ptr at some point $t' \in (t_{M@inv}, t]$. Since $B[j] \notin \mathcal{B}_i$, $B[j]$ is not in use during $I_i = [t_{2i}, t_{2i+1})$. Thus, p must read j from Ptr before t_{2i}, hence, we have $t' < t_{2i}$. Therefore, $t_{M@inv} < t_{2i}$. This is a contradiction, because $M \in \mathcal{O}_i$, and thus $t_{M@inv} \in [t_{2i}, t_{2i+1})$. □

Claim 6.22. *For all $i \in \{0, \ldots, k\}$, at point t_{2i}, all registers in $\mathcal{B}_i$ are in their initial states.*

Proof. All registers are in their initial states at t_0, so the claim is true for $i = 0$. Now fix some $i \in \{1, \ldots, k\}$. Suppose for the sake of contradiction that there is a register $B[j] \in \mathcal{B}_i$, such that $B[j]$ is not in its initial state at point $t_{2i} = t_{E_i@rsp}$. Since $B[j] \in \mathcal{B}_i$, index j is written into an entry of Ptr during `Reset()` operation call E_i. By the implementation, before that, in line 5 of an iteration of the loop in E_i, the calling process writes $\perp$ into $B[j]$ in line 4 of the same iteration of the loop in E_i. Let t_r be the point at which that $B[j]$.`Write(⊥)` is executed during E_i. Since the loop of E_i is executed in (t_{2i-1}, t_{2i}),

$$t_r \in (t_{2i-1}, t_{2i}). \tag{6.21}$$

Register $B[j]$ is not in its initial state at t_{2i}, only if some process p writes a non-$\perp$ value to $B[j]$ at some point during (t_r, t_{2i}). Process p can only write a non-$\perp$ value to register $B[j]$ in line 19 of an `RWrite()` call Op executed during a `TAS()` call M. Let $t_{Op@19}$ be the first point at which p writes to register $B[j]$ during (t_r, t_{2i}). Thus, we have

$$t_{Op@19} \in (t_r, t_{2i}). \tag{6.22}$$

By (6.21) and (6.22), we have $t_{2i-1} < t_{Op@19} < t_{2i}$.

At point $t_{Op@18}$, process p executes a C.`DRead()` call. If $t_{Op@18} \in [t_{2i-1}, t_{Op@19}) \subseteq [t_{2i-1}, t_{2i})$, then by Claim 6.17, M returns 1 at $t_{Op@18}$. This contradicts the assumption that p writes to $B[j]$ at $t_{Op@19} > t_{Op@18}$. Therefore, $t_{Op@18} < t_{2i-1}$. This implies that

$$t_{M@inv} < t_{2i-1}. \tag{6.23}$$

By our assumption, process p writes to register $B[j]$ at point $t_{Op@19}$, thus by Claim 6.19, no process writes $\perp$ into $B[j]$ in line 4 of a `Reset()` call during $[t_{M@inv}, t_{Op@19}]$, which by contains (6.23) during $[t_{2i-1}, t_{Op@19}]$. This is a contradiction, because a process executes a $B[j]$.`Write(⊥)` call in line 4 of E_i at t_r and by (6.21) and (6.22), $t_r \in [t_{2i-1}, t_{Op@19})$. □

Let Λ_i be the subtranscript of Λ that contains only events that occur during interval I_i, for $i \in \{0,\ldots,k\}$.

Claim 6.23. *History $\Gamma(\Lambda_i|\mathcal{O}_i)$ has a linearization S_i of a one-time TAS object, such that S_i contains all operation calls in $\mathcal{O}_i$.*

Proof. Let T_i be the subsequence of Λ that contains all invocation and response events of `TAS()` operation calls in $\mathcal{O}_i$ executed during I_i (i.e. $\Gamma(\Lambda_i|\mathcal{O}_i)$) as well as all operation calls on registers of B executed in those `TAS()` calls during I_i (i.e. $(\Lambda_i|\mathcal{O}_i)|B$). We first show that T_i is a transcript that can be obtained from the one-time TAS object O, where each register R_j is replaced with $B[Ptr[j]]$, for $j \in \{0,\ldots,m-1\}$.

By Observation 6.9, $C = 0$ throughout I_i. Therefore, each time a process executes C.`DRead()` from some `TAS()` operation call during this interval, its following if-condition evaluates to false. Thus, no TAS call that responds in this transcript returns in any of lines 7, 9, 11, 13, 16, 18 and 20. By Claim 6.21, operation calls in $\mathcal{O}_i$ access only registers of B that are in $\mathcal{B}_i$, and according to Claim 6.22, all registers of $\mathcal{B}_i$ are in their initial state at the beginning of I_i. By Claim 6.15(a) and Claim 6.13(a), $B[Ptr[j]] \neq B[Ptr[j']]$, for any two distinct indices $j, j' \in \{0,\ldots,m-1\}$, at any point during this transcript. Moreover, no `Reset()` call overlaps I_i, and so no process changes the value stored in an entry of Ptr in this transcript. Thus, T_i is a transcript that can be obtained from executing `TAS()` calls on O, where each R_j is replaced with $B[Ptr[j]]$, for $j \in \{0,\ldots,m-1\}$.

Let S'_i be the linearization of $\Gamma(T_i)$. Note that $S'_i \neq \emptyset$, for $i \in \{0,\ldots,k-1\}$, because E_i is executed by the process whose `TAS()` call completes during I_i, and so at least that call appears in S'_i. If $S'_k = \emptyset$ and $\mathcal{O}_k \neq \emptyset$, then no `TAS()` call in $\mathcal{O}_k$ completes during $I_k = [t_{2k}, t_{2k+1})$, so

they do not complete in Λ. In this case, we add the invocation event of the `TAS()` call with earliest invocation in $\mathcal{O}_k$ to S'_k, followed by a matching response with return value 0.

We construct a sequential history S_i from S'_i, and prove that S_i is a linearization of $\Gamma(\Lambda_i|\mathcal{O}_i)$ that contains all operation calls in $\mathcal{O}_i$: First all operation calls in S'_i are added to S_i in the same order as they appear in S'_i, and then all operation calls in $\mathcal{O}_i$ that do not appear in S'_i are added in the order of their response events.

All operation calls in $\mathcal{O}_i$ are invoked in I_i. Sequential history S'_i is a linearization of $\Gamma(T_i)$, and so it contains all operation calls in $\mathcal{O}_i$ that are invoked and respond during I_i, and it may contain some operation calls in $\mathcal{O}_i$ that are invoked but do not respond during I_i. This implies that S'_i, and thus S_i contains all complete operation calls in $\Gamma(\Lambda_i|\mathcal{O}_i)$. In addition to those, we add all operation calls in $\mathcal{O}_i$ that do not appear in S'_i to S_i. Hence, S_i contains all operation calls in $\mathcal{O}_i$ (and no other operation calls).

To prove that S_i is a valid history on a one-time TAS object, we need to show that only the first operation call in S_i responds 0, and all other operation calls return 1. As S'_i is a linearization of a history on a one-time TAS object, the first `TAS()` call that appears in S'_i returns 0, and all other operation calls in S'_i return 1. (In the case that the linearization of $\Gamma(T_k)$ is empty, we create S'_i so that it contains only one successful `TAS()` call.) All other operation calls in S_i (those that are not in S'_i) are the ones that are in $\mathcal{O}_i$, but do not complete during I_i. According to Claim 6.20(b), all such operation calls return 1. Thus S_i is valid.

Finally, for the following reasons the happens before order of operation calls in $\Gamma(\Lambda_i|\mathcal{O}_i)$ is preserved in S_i. First, S'_i is a linearization of $\Gamma(T_i)$, and so it preserves the happens before order of operation calls. Second, all operation calls that we add to S_i that are not in S'_i are pending at t_{2i+1}, and so adding them after operation calls in S'_i and according to the order of their response time preserves the happens before order of operation calls. In the case that the linearization of $\Gamma(T_k)$ is empty, all operation calls in $\mathcal{O}_i$ are pending at t_{2k+1}, therefore they can be ordered arbitrarily in S_i. $\square$

We can now complete the proof of Lemma 6.6. For convenience, we restate it here.

Lemma 6.6 (Restated)**.** *For any permissible transcript Λ^* on L, in which no two* `Reset()` *calls overlap, history $H = \Gamma(\Lambda^*)$ has a linearization S, satisfying:*

(a) all complete `Reset()` *calls in H linearize at their responses, and*

(b) if H contains a pending `Reset()`*, then it is the last operation call to linearize.*

Recall that we assume that Λ^* contains k `Reset()` calls $E_1,\ldots,E_k$, and all of them except possibly the last one complete in Λ^*. Also recall that we assumed that if E_k is pending at the end of Λ^*, we let E_k run to completion (which is possible because `Reset()` is wait-free), and Λ denote the resulting transcript, and otherwise we let $\Lambda = \Lambda^*$. Thus, it is enough to prove that $\Gamma(\Lambda)$ has a linearization, because then $\Gamma(\Lambda^*)$, which is a prefix of $\Gamma(\Lambda)$, also has a linearization.

We construct a sequential history S from $H = \Gamma(\Lambda)$ as follows: For each $i \in \{0,\ldots,k\}$, let S_i be the linearization of $\Gamma(\Lambda_i|\mathcal{O}_i)$ that contains all operation calls in $\mathcal{O}_i$, By Claim 6.23, such a linearization exists. We add all operation calls in S_0 to S in the same order as they appear in S_0. Next, for i from 1 to k, we repeat the following steps. We append all operation calls in $\mathcal{F}_i$ to S, ordered by their response time, and then we append `Reset()` call E_i, followed by all operation calls in $\mathcal{O}_i$, as they appear in S_i.

Next we prove that S is a linearization of H. Recall that for each $i \in \{0,\ldots,k\}$, $\mathcal{O}_i$ is the set of all `TAS()` operation calls invoked in $I_i = [t_{2i}, t_{2i+1})$, and for $i \in \{1,\ldots,k\}$, $\mathcal{F}_i$ is the set of all `TAS()` calls invoked in $[t_{2i-1}, t_{2i})$. Since $\left(\bigcup_{0\le i\le k}[t_{2i}, t_{2i+1})\right) \cup \left(\bigcup_{1\le i\le k}[t_{2i-1}, t_{2i})\right)$ covers the entire duration of H, the set of operations in $\left(\bigcup_{0\le i\le k}\mathcal{O}_i\right) \cup \left(\bigcup_{1\le i\le k}\mathcal{F}_i\right)$ contains all `TAS()` calls in H. Therefore by the construction, sequential history S contains all `TAS()` and `Reset()` calls in H.

The happens before order of operations is violated if we have $t_{M_1@rsp} < t_{M_2@inv}$ for two operation calls M_1 and M_2 in H, but M_2 precedes M_1 in S. By definition of $\mathcal{O}_i$ and $\mathcal{F}_i$, we have

$$inv(O_0) < inv(F_1) < rsp(E_1) < inv(O_1) < inv(F_2) < rsp(E_2) < \\ inv(O_2) < \cdots < inv(O_{k-1}) < inv(F_k) < rsp(E_k) < inv(O_k), \tag{6.24}$$

where $O_i \in \mathcal{O}_i$ and $F_i \in \mathcal{F}_i$, for $i \in \{1,\ldots,k\}$. Consider two operation calls M_1 and M_2 in H, such that $t_{M_1@rsp} < t_{M_2@inv}$. The happens before order of M_1 and M_2 is preserved in S, because,

(a) if $M_1, M_2 \in \mathcal{O}_i$, for some $i \in \{0,\ldots,k\}$, then they are ordered based on their order in S_i which preserves the happens before order of operation calls in $\Gamma(\Lambda_i|\mathcal{O}_i)$,

(b) if $M_1, M_2 \in \mathcal{F}_i$, for some $i \in \{0,\ldots,k\}$, then they are ordered by their response time,

(c) if M_1 and M_2 are both `Reset()` calls, then they do not overlap and they appear in S in the same order as in Λ,

(d) otherwise, by (6.24) and the construction of S, M_1 precedes M_2 in S.

For each $i \in \{0,\ldots,k\}$, sequential history S_i is a valid history on a one-time TAS object, thus the first `TAS()` operation call in S_i returns 0, and all other operation calls in S_i return 1. Moreover, by Claim 6.20(a) all operation calls in $\mathcal{F}_i$ return 1. Therefore, in S, the first `TAS()` operation call and each `TAS()` operation call that immediately follows a `Reset()` call return 0, and all other `TAS()` calls return 1. Thus, history S is a valid history on a long-lived TAS object. Hence, S is a linearization of H.

For any $i \in \{1,\ldots,k\}$, any `TAS()` call that is invoked before $t_{E_i@rsp} = t_{2i}$ is in $\left(\bigcup_{0\le j<i} \mathcal{O}_j\right) \cup \left(\bigcup_{1\le j\le i} \mathcal{F}_j\right)$, and therefore appears before E_i in S, for any $i \in \{1,\ldots,k\}$. Thus, S can be obtained from H by assigning appropriate linearization points, where each `Reset()` call is linearized at its response. This proves Part (a) of Lemma 6.6.

Moreover, if E_k is pending in Λ^*, then the response of E_k is the last event in Λ. Therefore, $\mathcal{O}_k$ is empty, because no `TAS()` call is invoked after E_k in Λ. Thus by the construction of S, E_k appears last in S in this case, which proves Part (b) of Lemma 6.6. This concludes the proof of Lemma 6.6.

6.4.2 Proof of Theorem 6.1

Consider a one-time TAS implementation O from m registers, and let L be the long-lived TAS implementation of Figures 6.2 and 6.3 that is obtained from O. By Corollary 6.8, L is linearizable. To implement B, Ptr, $Annc$, and data structures of the `recycle()` operation, $O(n+m)$ registers are used. The ABA-detecting register C can be implemented from $O(n)$ registers (see Theorem 5.1 of Chapter 5). Each register of L stores a value in $\{0,\ldots,2n+m\} \cup \{\perp\}$, or a value that a register of O is required to hold, which by the assumption is at most b bits. Thus, $O(n+m)$ registers, each of size $\max\{\log(2n+m+2), b\}$ bits, suffice to implement L from O.

A `TAS()` operation call of L executes all steps of a `TAS()` call of O, and with each of those steps it incurs constant additional steps, hence, it has asymptotically the same step complexity. Finally, a `Reset()` operation call of L includes m repetitions of a constant number of steps, and so it requires $O(m)$ steps in the worst case.

Chapter 7

Taggable Objects

Many shared memory algorithms benefit from a standard technique called *tagging*, where registers or other shared objects get augmented with additional values, called tags. In a system, with resources of unbounded size, a process can generate a new tag by incrementing a local variable c, and then using its id and value c as the tag. Some applications of tagging are ABA detection, implementation of concurrent data structures, and memory management. Researchers have been introducing ad-hoc techniques to enable bounded tagging for their algorithms.

This chapter presents a systematic investigation of the tagging problem, by introducing new types that maintain a fixed finite pool of tags. A process can efficiently find a free tag from that pool and communicate it to other processes by storing it in a shared object. Other processes can obtain references to tags by reading them from shared objects, and later release those references via a dedicated operation. The main safety property provided by our abstraction is that whenever a process obtains a new tag from the pool, then immediately before that the tag was free, that is, no process had a reference to it and it is not stored in elsewhere than in the pool. New tags can be taken from the pool infinitely many times, as long as the algorithm that uses this abstraction guarantees that the number of tags to which processes have references, is bounded by some parameter τ that is much smaller than the size of the pool.

Often, tags need to augment other data that is written to objects. We present two types, that mainly differ in what operations can be used to store or retrieve (data,tag) pairs. In the first type, called *taggable register array* (TRA), these operations are reads and writes. The second type, called *taggable LL/SC array* (TLSA), supports load-linked/store-conditional (LL/SC) operations. We provide a specification of these types in Section 7.1 followed by simple applications to motivate them in Section 7.2. We present an implementation of TLSA from LL/SC objects and registers,

and an implementation of TRA from registers in Section 7.3. Each of those implementations is wait-free, all operations have constant step complexity, shared base objects have bounded size (typically it is logarithmic in the number of processes), and the number of base objects used is bounded (typically polynomially in the number of processes). Finally in Section 7.5, we prove that both implementations are correct.

7.1 Taggable Objects Specification

This section provides a formal specification of two types, taggable register array (TRA) and taggable LL/SC array (TLSA). Each of these is instantiated by two parameters, m and τ, and maintains an array $A[0 \ldots m-1]$ of size m. The parameter τ is, roughly, a bound on the number of tags that can be simultaneously referenced by processes (a formal definition will be provided below). To specify the values of the parameters m and τ, we will sometimes write (m,τ)-TRA and (m,τ)-TLSA. The tags come from the domain $T = \{0,\ldots,\Delta(m,n,\tau)-1\}$, for some function $\Delta : \mathbb{N} \to \mathbb{N}$ that is specified based on the implementation of the type. (In our implementations, $\Delta = O(m^2n^6 + n^2\tau^2 + mn^4\tau)$; see Section 7.2.4 for more details.) The components of the array A behave essentially as registers for TRA, and as LL/SC objects for TLSA, but they provide additional functionality. Each array component can be used to store data augmented with a tag, and a retrieval of a (data,tag) pair from one array component automatically prevents the retrieved tag from being freed.

In the following, we give a specification of the types TLSA and TRA. In particular, we describe how the objects behave if their operations are executed only sequentially. Both types support the operations `GetFree()` and `Release()`.

(S1) Operation `GetFree()` returns a *free* tag g from pool T, and as a result g is **active**.

We will define later precisely what it means that a tag is free.

(S2) Operation `Release(`g`)` takes as parameter a tag g that is active, and as a result g is *inactive*.

(Even when a tag has been released, it may not be free because other processes may still need access to it, as explained below.) Thus, a tag is **active** if a `GetFree()` returned g and was not followed by a `Release(`g`)`.

Once a tag has become active through a `GetFree()` operation, processes can augment it with data values, and communicate the resulting (data,tag) pairs via the shared objects of array A. In case of type TRA, this is done with the operations `TWrite()` and `TRead()`. Specifically,

(S3a) operation `TWrite(`$i,(x,g)$`)` writes the pair (x,g) consisting of a data value x and a tag g to $A[i]$, and

(S4a) operation `TRead(`i`)` returns the pair stored in $A[i]$.

Similarly, in case of type TLSA, tags can be stored in and retrieved from A using operations `TLL()` and `TSC()` that behave like `LL()` and `SC()` operations on LL/SC objects:

(S3b) Operation `TSC(`$i,(x,g)$`)` attempts to write the pair (x,g) to $A[i]$, and returns a boolean value indicating whether the write succeeded or not (if it does not succeed, $A[i]$ remains unchanged). Operation `TSC(`$i,(x,g)$`)` by p succeeds if and only if p previously executed `TLL(`i`)`, and since then no other successful `TSC(`$i,\cdot$`)` operation was performed.

(S4b) Operation `TLL(`i`)` returns the current pair stored in $A[i]$.

As a result of a `TLL()` or a `TRead()` that returns a pair (x,g), a process has tag g *protected*, and can from then on safely use that tag. (The meaning of safe is explained later.) Since multiple `TLL()` or `TRead()` operations may return the same tag, a tag may be protected multiple times. We define a function $Protected(g,p)$ that counts the number of times tag g is protected by p. A process *protects* a tag if $Protected(g,p) = k \geq 1$.

(S5) If $Protected(g,p) = k$ before a `TLL()` or `TRead()` operation call by p that returns tag g, then $Protected(g,p) = k+1$ after this operation call.

(S6) If $Protected(g,p) = k$ before an `Unprotect(`g`)` call by p, then $Protected(g,p) = k-1$ after this operation call.

(The constraints on the use of these types, which are being introduced later, will prevent $Protected(g,p)$ from becoming negative.)

The goal of these operations is that access to tags is always safe, in the sense that no process could be poised to use a tag g, when a `GetFree()` operation call returns g. To make this precise, we distinguish between *free* and *occupied* tags.

A tag g is **occupied** if

- it is active; or
- some process protects it; or
- a pair $(\cdot,g)$ is stored in some element of array A.

A tag g is called **free** if it is not occupied. The *safety property* guaranteed by types TLSA and TRA is that just before a `GetFree()` operation returns a tag, that tag is free (as stated in **(S1)**). A summary of TRA and TLSA specifications is provided in Figure 7.1.

Safety can, of course, only be ensured if processes access tags *properly*. Any algorithm using these primitives must satisfy the following *constraints* in order for the primitive to behave as in its specification. In particular, for any process p,

(C1) when p calls `Unprotect(`g`)`, p protects tag g,

(C2) when p calls `Release(`g`)`, g is active, and

(C3) when p calls `TSC(`$\cdot,(\cdot,g)$`))` or `TWrite(`$\cdot,(\cdot,g)$`))`, g is occupied.

Moreover, in order to not run out of free tags, the algorithm using an (m,τ)-TRA or an (m,τ)-TLSA must ensure that

(C4) there are never more than τ tags active or protected (where $\tau < \Delta(m,n,\tau)$).

$$
\begin{aligned}
Free(g) &\stackrel{Def}{\equiv} \neg Occupied(g) \\
Occupied(g) &\stackrel{Def}{\equiv} Active(g) \vee Protected(g) \vee Stored(g) \\
Active(g) &\stackrel{Def}{\equiv} \text{ tag } g \text{ is active} \\
Protected(g) &\stackrel{Def}{\equiv} \exists p \in \{0,\ldots,n-1\} : Protected(g,p) > 0 \\
Stored(g) &\stackrel{Def}{\equiv} \exists i \in \{0,\ldots,m-1\} : A[i] = (\cdot, g)
\end{aligned}
$$

Operation `GetFree()`

Precondition: $\exists g : Free(g)$.
Postcondition: $Active(g)$, and returns g.

Operation `Release(`g`)`

Precondition: $Active(g)$.
Postcondition: $\neg Active(g)$.

Operation `Unprotect(`g`)`

Precondition: $Protected(g,p) = k \geq 1$.
Postcondition: $Protected(g,p) = k - 1$.

Operation $\texttt{TRead}_p(i)$

Precondition: $A[i] = (x,g)$, and $Protected(g,p) = k \geq 0$.
Postcondition: $Protected(g,p) = k + 1$, and returns (x,g).

Operation $\texttt{TLL}_p(i)$

Precondition: $A[i] = (x,g)$, and $Protected(g,p) = k \geq 0$.
Postcondition: $Protected(g,p) = k + 1$, p has a valid link to $A[i]$, and returns (x,g).

Operation $\texttt{TWrite}_p(i,(x,g))$

Precondition: $Occupied(g)$.
Postcondition: $A[i] = (x,g)$ and $Stored(g)$.

Operation $\texttt{TSC}_p(i,(x,g))$

Precondition 1: $Occupied(g)$, and p has a valid link to $A[i]$.
Postcondition 1: $A[i] = (x,g)$, $Stored(g)$, invalidates all processes' links, and returns `true`.
Precondition 2: $Occupied(g)$, and p does not have a valid link to $A[i]$.
Postcondition 2: Returns `false`.

Figure 7.1: Specification of a TRA/TLSA object A

Processes can achieve **(C4)** by releasing and unprotecting sufficiently many tags. For example, in one of the applications presented in the following section, we choose $\tau = n$, because at any point each process can have at most one tag protected or active.

Note that `Release`(g) is the counterpart of `GetFree()`, and `Unprotect`(g) is the counterpart of `TRead()` or `TLL()`. But if a process begins to protect a tag g as a result of a `TRead()` or `TLL()` operation call, it needs to call `Unprotect`(g) itself, while any process can call `Release`(g) for an in-use tag g, no matter which process received g from a `GetFree()` call. This property will allow us later (in Section 7.2.3) to design an extension of this type that can replace other memory reclamation techniques, such as Hazard Pointers (Michael, 2004b) or Pass-the-Buck (Herlihy, Luchangco and Moir, 2002), for list based data structures.

For the ease of description, in the rest of this chapter (unless it is said otherwise and except for the proofs), we adopt the following conventions. First we ignore the data values that are augmented by tags when storing or loading them from taggable arrays. Moreover, an (m,τ)-TLSA or (m,τ)-TRA object X for $m = 1$ maintains a taggable array of size 1. Therefore, the first parameter of all operation calls on X is always 0. Hence, we omit that parameter if it is clear from the context that we refer to such an object with $m = 1$. So for example, if X is a $(1,\tau)$-TLSA object, we write X.`TLL()` instead of X.`TLL(0)`, X.`TSC`(g) instead of X.`TSC`$(0,(\cdot,g))$, and X instead of $X[0]$.

7.2 Applications and Results

To illustrate the advantages of our abstraction, we first describe some simple applications in Sections 7.2.1 and 7.2.2. Then, we extend the specification of taggable objects in Section 7.2.3, so that it can be used as a memory management scheme for linked data structures. Section 7.2.4 gives a summary of the results.

7.2.1 Round-Based Algorithms.

In a recent mutual exclusion algorithm (Giakkoupis and Woelfel, 2014), processes make multiple attempts to enter the critical section. In each such attempt, a process p increments a round number, c, and when it writes to some objects, it writes pairs of the form (x,c). (For simplicity, the algorithm uses a CAS object, which as argued in the paper, can be replaced by registers without affecting the efficiency of the mutual exclusion algorithm.) The only purpose of writing round numbers is that if some other process reads different pairs, say (x,c) and (x',c') written by p, then it can decide by comparing c and c', whether those pairs were written by p in the same or in different rounds. The temporal order relation between rounds is irrelevant. Since there is no bound on the number of attempts a process makes, correctness can only be ensured with this implementation if unbounded registers are used to store the round numbers. Therefore, the authors describe an ad-hoc way of recycling round numbers in that particular algorithm, but no detailed correctness proof is given. With our abstraction, this becomes trivial: We can use an (m,τ)-TRA object, where m is the number of registers that need to store round numbers in the algorithm, and $\tau = cn$, for a large enough integer c. At the beginning of a round, a process calls `GetFree()`, and uses the returned tag, g, as its round number. During the round, it uses the operations `TWrite()` and `TRead()` provided by the TRA object whenever it wants to write or read a value from a register that may store a round number (tag). Once a process does not need the tag value g that it read from an element of the taggable array, it unprotects tag g with an `Unprotect(`g`)` call, and at the end of the round it releases its own round number by calling `Release(`g`)`.

7.2.2 Pointer Swinging

Many shared memory algorithms are based on the following template: There is a pointer X, that points to a block of objects that store the current state of the data structure. A process may modify the data structure as follows: First it allocates a new block, B, of objects. Then it reads

X and the data structure from the block pointed to by X, computes the new state of the data structure, and writes the new state to B. Finally, the process tries to change X so that it points to B. If this attempt fails, then the process has to start over.

When X is a CAS (or an LL/SC) object, the algorithm that uses this approach belongs to an algorithmic class called *single compare-and-swap universal (SCU)* (Alistarh, Censor-Hillel and Shavit, 2016). This provides a simple transformation of any sequential data structure to a concurrent lock-free one. Many lock-free data structures (Clements, Kaashoek and Zeldovich, 2012; Michael and Scott, 1996), and the *read-copy-update (RCU)* mechanism employed by the Linux kernel (McKenney and Slingwine, 1998) belong to this class.

Some algorithms based on the template above need only a register for X. Examples are the implementation of a CAS object from name consensus and registers (Golab, Hadzilacos, Hendler and Woelfel, 2012), and constructions to augment a wide class of objects with wait-free reset or write operations (Aghazadeh, Golab and Woelfel, 2013, 2014; Aghazadeh and Woelfel, 2014), as discussed in Chapters 4 and 6.

Under the assumption that there are an unbounded number of blocks available, implementing such algorithms is trivial. However, implementations in bounded space are not straightforward, and it is not surprising that a significant amount of the technical work in those constructions is devoted to ad-hoc memory reclamation.

Our taggable array types provide an abstraction that allows an elegant solution to the memory reclamation problem encountered in those algorithms: For X we use either a TRA or a TLSA object. Each tag g is associated with a block $B[g]$ of memory, and X stores a tag that is an index of the current block. To "allocate" a new block, a process calls `GetFree()` to obtain an index g' of a free block. After successfully updating X to g', the process calls `Release(`g'`)` to indicate that it is not using block $B[g']$ anymore. The safety property of the taggable array primitive guarantees that no `GetFree()` returns an index to an occupied block, that is, one that a process may be about to change. In the following, we provide an example to show how a TLSA

```
// X is LL/SC object.
// B is an array of unbounded size.

Operation Update(x)
1  g' := an index into B that has never been used
2  repeat
3      g := X.LL()
4      b := B[g]
5      Determine new state from b and x; write to B[g']
6  until X.SC(g')
7  return b

Operation Read()
8  g := X.LL()
9  b := B[g]
10 return b
```

Figure 7.2: SCU with Unbounded Memory

```
// X is a (1,2n)-TLSA object.
// B is an array of size Δ(1,n,2n).

Operation Update(x)
11 g' := X.GetFree()
12 repeat
13     g := X.TLL()
14     b := B[g]
15     Determine new state from b and x; write to B[g']
16     X.Unprotect(g)
17 until X.TSC(g')
18 X.Release(g')
19 return b

Operation Read()
20 g := X.TLL()
21 b := B[g]
22 X.Unprotect(g)
23 return b
```

Figure 7.3: SCU with Bounded Memory

can be used to implement a general SCU algorithm, and another example in which a TRA is used to implement a k-word register.

Single Compare-and-Swap Universal. We can imagine an SCU algorithm which implements a lock-free data structure to include two main operations: `Update()` and `Read()`. Any operation that modifies the object can be implemented using a lock-free `Update(x)` operation. `Update(x)` modifies the object based on some parameter x (e.g. x can be (`Insert`,10) for a tree data structure in order to insert a node with key value 10), and returns its previous value. Any read-only operation can be implemented using the wait-free `Read()` operation, which returns the object's value. Figure 7.2 depicts a general SCU algorithm if unbounded memory is available.

Any `Read()` operation linearizes with the load of X, and an `Update()` call linearizes with the successful X.`SC()` call.

Memory reclamation for SCU can be achieved with techniques such as Hazard Pointer (Michael, 2004b) or Pass-the-Buck (Herlihy, Luchangco and Moir, 2002). However applying these techniques makes the `Read()` operation *lock-free*, because overlapping `Update()` calls may force a reader to repeat the steps of its `Read()` call infinitely many times. Moreover, memory allocation must be provided by the system when adopting these techniques.

Figure 7.3 presents a solution with bounded memory using a $(1,2n)$-TLSA, in which the `Update()` is still lock-free, but the `Read()` operation is wait-free. In this implementation $\tau = 2n$. Since each process can have at most two tags either protected or active while its `Update(`x`)` or `Read()` operation is pending, and none otherwise, constraint **(C4)** is satisfied. Moreover, a process only calls `Unprotect(`g`)` if it protects g, and `Release(`g`)` if the tag is active. Finally, `TSC(`g`)` is only called when g is active. Hence, the algorithm of Figure 7.3 also satisfies all **(C1)**-**(C3)**. Thus, the safety property of TLSA ensures that when a `GetFree()` returns some tag g' at some point, then g' is not occupied at that point. Therefore, no other process has a pending operation in which it has loaded tag g' from X or received g' from a `GetFree()` at this point.

Multi-Word Registers. Another example of the pointer swinging technique is the linearizable k-word register implementation in Figure 7.4 from single-word registers and a $(1,n)$-TRA object. As we will show in Section 7.3, a $(1,n)$-TRA object can be constructed from polynomially many single-word registers such that all operations on the TRA object have constant step complexity. Therefore, each operation of Figure 7.4 executes k steps on single-word registers and at most 3 steps on the TRA object. Hence, this implementation has asymptotically optimal step complexity $O(k)$, and the space requirement is polynomial in n and k. A more space efficient solution that uses only $O(n^2k)$ single-word registers and has $O(k)$ step complexity was given in Chapter 4. That algorithm is quite involved; the purpose of our example here is to show that the problem

```
// X is a (1,n)-TRA object.
// B_i for i ∈ {1,...,k} is an array of Δ(1,n,n) single-word registers.
```

Operation `Write`$(x_1,\ldots,x_k)$	**Operation** `Read()`
24 $g := X$.`GetFree()`	**29** $g := X$.`TRead()`
25 **for** $j = 1,\ldots,k$ **do**	**30** **for** $j = 1,\ldots,k$ **do**
26 $\quad B_j[g] := x_j$	**31** $\quad x_j := B_j[g]$
27 X.`TWrite`(g)	**32** X.`Unprotect`(g)
28 X.`Release`(g)	**33** **return** $(x_1,\ldots,x_k)$

Figure 7.4: A k-Word Register Implementation

has an almost trivial solution, when developed using our TRA abstraction.

The `Read()` operation linearizes with its X.`TRead()` call, and the `Write()` operation linearizes with its X.`TWrite()` call. In this implementation, $\tau = n$. Since at each point, exactly one tag is protected or active per each pending `Read()` or `Write()` operation call, condition **(C4)** is satisfied. Moreover, conditions **(C1)**-**(C3)** are also satisfied, because a process calls `Unprotect`(g) only when it protects g, and calls `Release`(g) and `Write`(g) only when tag g is active. Thus the safety property of TRA guarantees that when a `GetFree()` call executed during a `Write()` operation W returns g', this tag is not protected, or active, or stored in X. Therefore, no process accesses the registers corresponding to tag g' after this point and before g' is written to X at the linearization point of W.

Stack Implementations. Another example is that of a stack implementation with a wait-free `Peek()` operation. Michael (2004b) applied his Hazard Pointer technique to a lock-free stack implementation based on the IBM FreeList algorithm (IBM, 1983). The `Peek()` operation in the implementation by Michael (2004b) is lock-free. Because reading and announcing the index of the top node are two separate steps, so overlapping `Push()` and `Pop()` operations may force the process to keep repeating these steps infinitely many times. Therefore, it cannot easily be made wait-free. We can replace Hazard Pointers by using a TLSA object *Top* to store the address of the top element of the stack. As a result, a *Top*.`TLL()` operation not only returns a reference to that top element, it also protects it. This way, as we show in the following, it is straight-forward

```
// Top is a (1,k+n)-TLSA object.
// Node is an array of Δ(1,n,k+n) nodes.
// Initially, Top stores tag 0, which represents the bottom of the stack and will never
   be freed.

Operation Push(x)
34 new := Top.GetFree()
35 Node[new].Data := x
36 while true do
37     t := Top.TLL()
38     Node[new].Next := t
39     Top.Unprotect(t)
40     if Top.TSC(new) then return

Operation Peek()
41 t := Top.TLL()
42 if t = 0 then
43     Top.Unprotect(t)
44     return "empty stack"
45 data := Node[t].Data
46 Top.Unprotect(t)
47 return data

Operation Pop()
48 while true do
49     t := Top.TLL()
50     if t = 0 then
51         Top.Unprotect(t)
52         return "empty stack"
53     next := Node[t].Next
54     if Top.TSC(next) then
55         data := Node[t].Data
56         Top.Unprotect(t)
57         Top.Release(t)
58         return data
59     else
60         Top.Unprotect(t)
```

Figure 7.5: A stack Implementation with `Peek()` in Constant Step Complexity

to obtain a wait-free `Peek()` operation.

Figure 7.5 shows a stack implementation based on the IBM FreeList algorithm (IBM, 1983). Here we assume that the number of elements that can be stored in the stack is bounded by some value $k-1$, but this assumption is not in the original algorithm and the one with Hazard Pointers (Michael, 2004b). This implementation uses a $(1,k+n)$-TLSA object *Top*, and an array *Node* of $\Delta(1,n,k+n)$ nodes is used, where each node stores a data item *Data* and a next pointer *Next*. The `Push()` and `Pop()` algorithms are based on the ones given by Michael (2004b), but instead of protecting the top element with Hazard Pointers, this element is automatically protected by the TLSA object *Top*.

Initially, a dummy node $Node[0]$ (the node corresponding to tag 0) is stored in the list, which represents the bottom of the stack. This tag always remains active, and will never be freed. In order to push a new node into the stack, a process p calls `GetFree()` to get a tag corresponding

to a free node. Then p loads the tag t corresponding to the current top of the stack, by executing Top.`TLL()`, and writes it into the next pointer of the new node. Process p then unprotects t, because p is not going to read from or write to the node with index t before it starts another operation. Finally, process p tries to swing the pointer, by writing the new tag into Top with a Top.`TSC()` operation. If that `TSC()` succeeds, p's `Push()` operation is complete, and it linearizes with this successful `TSC()` call. Otherwise, another process modified Top, and so p repeats all the steps starting from Top.`TLL()` operation until its `TSC()` succeeds.

To pop from the stack, a process p loads tag t that corresponds to the top node by executing Top.`TLL()`. If the stack is empty, then p reads 0. Hence, since p is not going to access $Node[0]$ before it starts another operation, it unprotects this tag, and returns "empty stack". In this case, this operation linearizes with the last `TLL()` call. Otherwise, process p reads tag $next$ corresponding to the next node in the stack from $Node[t].Next$. Then, p tries to write $next$ into Top, in order to swing the pointer to the second top element in the stack. If it succeeds, then it unprotects and releases tag t (which corresponds to the old top node), and returns the data value stored in $Node[t]$. This way, once no process is protecting t anymore, t can be reused. This successful `TSC()` call is the linearization point of this `Pop()` operation. If p's `TSC()` fails, then some other process must have updated the top element of the stack, so p unprotects t (because its earlier `TLL()` operation which returned t also makes p protect this tag), and starts over.

Recall that Top.`TLL()` not only returns the tag value stored in Top, but it also protects this tag. As a result, it is now easy to implement a wait-free `Peek()` operation: Process p reads and protects the index (tag value) t of the top node of the stack in one step by executing Top.`TLL()`. If $t = 0$, then p returns "empty stack", and if not, P returns the data value stored in $Node[t]$. In either case, p also has to unprotect t before it returns, because p is not going to read from or write to the node at index t, before reading Top again. The `Peek()` operation linearizes with its `TLL()` call.

In this implementation, $\tau = k + n$, and that satisfies condition **(C4)**, for the following reasons:

First, only tags that correspond to nodes that are currently stored in the stack are active. Hence, at most k tags (including 0) are active. Also, each process can protect at most one tag and only while it has a pending operation. Therefore, in total at most $k+n$ tags are active or protected.

Moreover, when a process p calls `Unprotect(`g`)` (in lines 39, 43, 46, 51, 56 and 60), its last *Top*.`TLL()` call has returned g, and no `Unprotect(`g`)` has followed since that *Top*.`TLL()` call. Thus, g is protected by p when this process calls `Unprotect(`g`)`, so **(C1)** is satisfied. A process obtains tag g by loading *Top* before it calls `Release(`g`)`. So g is the tag corresponding to the node on top of the stack when `Release(`g`)` is called. Since any tag that corresponds to a node in the stack is active, g is active when `Release(`g`)` is called, so **(C2)** is satisfied. Similarly, a process calls `TSC(`g`)` if it reads g either from *Top*, or from *Next* of the node whose tag is stored in *Top*, which corresponds to a node in the stack. Therefore, g is occupied while a process executes a `TSC(`g`)`. Thus, **(C3)** is also satisfied. Hence, the safety property of TLSA ensures that when `GetFree()` returns g, no process accesses the node that corresponds to g, before g is written to *Top*, and so not before the node is added to the stack.

In this example, we only need to protect the top node of the stack, that is, the head of the linked list that implements the stack. However, for a general linked data structure, to read a node N, the process needs to protect the tag corresponding to N, as well as any tag(s) stored in N which represent node(s) referenced by N. In the following section, we explain how we can extend our specification to enable memory management for such data structures.

7.2.3 Extended Specification and Memory Management

With the specification described in Section 7.1, our taggable objects cannot easily be used for memory reclamation in linked data structures. Consider a linked list implementation for instance. A natural way to attempt to use our primitives would be to maintain a taggable array X of size m, and to associate a tag with every node that may be used in the list. This way, with a `GetFree()` operation, we get a tag corresponding to a free node. Let $Node[u]$ denote the node corresponding to tag u.

For a process p to search in the list, it reads some tag from the head pointer which is an entry of the taggable array X. Then p walks through the list by following $Next$ pointers. Suppose p reads some tag v from $Node[u].Next$, but before p reads $Node[v].Next$, another process removes $Node[v]$. Process p does not have v protected, so v may be freed and possibly reused in another part of the list.

To tackle cases like this, ideally we would like to ensure that by reading v from $Node[u].Next$, tag v becomes automatically protected. We can achieve this with our taggable array primitive only if each node is a component of the taggable array X. Thus, the total number of nodes in the system must be the same as the size, m, of the taggable array X. Since we already assumed that each node corresponds to a tag, number of tags in the system, $\Delta(m,n,\tau)$, is also the same as the size, m, of the taggable array. But at least with our implementation of taggable arrays, this is not possible, because the size of the tag domain, $\Delta(m,n,\tau)$, is much larger than m.

To deal with this problem, it is possible to extend our types, so that our objects can be used in a similar way as Hazard Pointer (Michael, 2004b) and Pass-the-Buck (Herlihy, Luchangco and Moir, 2002) for linked data structures. To do that, we introduce two operations: `Protect()` and `CancelProtect()`, and we allow tags that are active to be communicated to other processes not only through the taggable array X, but also through other objects in the system. Before we formally specify these operations, we go back to our linked list example, to motivate their formal specification. Suppose again that p reads v from $Node[u].Next$. Then the process immediately calls `Protect(`v`)`. The hope is that this `Protect()` call actually protects this tag. The difficulty is that the read of v from $Node[u].Next$ and the `Protect(`v`)` call are not executed in one atomic step. So it is possible that another process removes $Node[v]$ from the list before p manages to call `Protect(`v`)`. We do not know any way of implementing `Protect()` in such a way that it has a return value that indicates if the call was executed early enough or not, and so if the `Protect()` call was successful or not. However, we provide the following guarantee on when a `Protect(`g`)` call successfully protects g (a formal statement comes later): If g is occupied at

a point at or after the execution of `Protect(`g`)` by p, then p protects tag g from this point onwards. In our linked list example, after reading v from $Node[u].Next$ and calling `Protect(`v`)`, process p would read $Node[u].Next$ again. If it reads the same tag v, then v is occupied at the point of the second read, and so it is guaranteed that p now protects v. Thus, p can now read $Node[v].Next$ knowing that v is protected.

However, this comes at a price. A process cannot use the taggable array after a `Protect(`g`)` call, if it is unsure whether g was occupied at some point at or after its `Protect(`g`)` call. For that reason, we allow a process to cancel the potentially unsuccessful `Protect(`g`)` call, before executing any other operation on the taggable array. A process can do that by executing a `CancelProtect(`g`)` operation. So in our example, after process reads $Node[u].Next$ for the second time, if it reads the same tag v, then it proceeds with the rest of its operation, otherwise, it calls `CancelProtect(`g`)`.

Now we formally define our extension to TRA and TLSA types (as originally specified on page 151). The extended specification has two additional operations, `Protect()` and `CancelProtect()`, each takes as argument a tag g, and returns nothing, and has the following properties.

(S7) A `Protect(`g`)` call executed by process p at point t *succeeds* and increments the value of $Protected(g,p)$, at the first point at or after t, at which tag g is occupied, and before p has called `CancelProtect(`g`)`, if such a point exists. A `Protect(`g`)` call by p gets *cancelled* by a subsequent `CancelProtect(`g`)` call by p.

(S8) $Protected(g,p) = k$ after a `CancelProtect(`g`)` call, if $Protected(g,p) = k$ immediately before the last `Protect(`g`)` call by p preceding this `CancelProtect(`g`)`.

In order to ensure the safety property, that is, that a `GetFree()` returns a free tag, in the extended TLSA and TRA types, the following constraints must be met by the algorithm that uses the TRA or the TLSA object:

(C5) After a `Protect(`g`)` call by p, process p is not allowed to execute any operation other than `CancelProtect(`g`)` on the TRA or TLSA object until either that `Protect(`g`)` call succeeded, or has been cancelled by p.

(C6) Process p can execute `CancelProtect(`g`)` only after a `Protect(`g`)` call by p and before p executes any other operation on the TRA or TLSA object.

(C7) If a process p calls `Protect(`g`)`, then g must have been occupied at some point since p's preceding `Protect()` call or the beginning of the execution, whichever comes last.

Constraint **(C7)** is required in order to ensure implementations with bounded space, because otherwise a process could prevent another process to find a free tag infinitely many times. It is often straightforward to guarantee this condition for linked data structures. One option is to ensure the following invariant for any node $Node[g]$ (the node that corresponds to tag g): "If $Node[g]$ stores a tag g' (e.g., g' could be the address of the next node following $Node[g]$ in a linked list) and g is occupied, then g' is also occupied." Then, assuming a process p protects g whenever it reads any fields of $Node[g]$, Property **(C7)** is achieved by calling `Protect(`g'`)` immediately after reading g' from $Node[g]$: At the point of that read, g' is occupied because g is, and this read is followed by `Protect(`g'`)` call.

This invariant will be naturally satisfied for typical implementations of linked data structures: To create a new node, a process would call `GetFree()`, and only when it removes a node with an address corresponding to a tag g from the data structure, the process would call `Release(`g`)`. So any node reachable in a data structure is active, and so is occupied.

7.2.4 Results

We present in Section 7.3 implementations of extended taggable LL/SC and extended taggable register arrays with each operation having constant step complexity, and using bounded space. In the remainder of Chapter 7, when we refer to type TRA or TLSA, we mean the extended specification of that type, as discussed in Section 7.2.3. All our results and proofs hold for this

extended specification, but presumably better space bounds could be achieved for the restricted specification.

Theorem 7.1. *Let $M(m,n,\tau)$ and $\Delta(m,n,\tau)$ be sufficiently large polynomials. For each integer b, there are linearizable implementations of type (m,τ)-TRA from $M(m,n,\tau)$ registers and of type (m,τ)-TLSA from $M(m,n,\tau)$ registers and LL/SC objects, such that each operation has constant step complexity, each taggable array entry stores a b-bit value in addition to a tag, and the base objects have size $O(b+\log(\Delta(m,n,\tau)))$ bits.*

In Section 7.5.10, we show that the following upper bounds hold for M and Δ:

$$M(m,n,\tau) = O(mn^5 + n^3\tau), \text{ and} \tag{7.1}$$

$$\Delta(m,n,\tau) = O(m^2n^6 + n^2\tau^2 + mn^4\tau). \tag{7.2}$$

(We believe that it is possible to reduce these values, but doing so would make the algorithms more complicated.) For instance, for $\tau = O(n)$ and $m \geq 1$, we have $\Delta(m,n,\tau) = O(m^2n^6)$.

An LL/SC object can be implemented from a single compare-and-swap (CAS) object and $O(n)$ registers, in such a way that each `LL()` and each `SC()` operation has constant step complexity (Jayanti and Petrovic, 2003). Therefore, we can implement a TLSA object from CAS objects and registers.

In the following two sections, we describe the implementations of TLSA and TRA objects. Section 7.5 provides correctness proofs for both implementations.

7.3 TRA and TLSA Implementations

This section presents a high level explanation of the main ideas in our implementations of *extended* TRA and TLSA types. A more detailed description is provided in Section 7.4. These algorithms consist of two parts: the first is organizing and managing tags (Section 7.3.1), and

the second is reading/loading and writing/storing (data,tag) pairs into elements of the taggable array (Sections 7.3.2 and 7.3.3).

7.3.1 Managing Tags

The part that is responsible for dealing with tags in both TRA and TLSA implementations is provided in Figure 7.6. Tags are partitioned into βn *blocks* $b_0,\ldots,b_{\beta n-1}$, and each block contains δ tags. Thus, $\Delta(m,n,\tau) = \delta \cdot \beta \cdot n$. Each block is *owned* by exactly one of the n processes, and each process owns β blocks.

A process' `GetFree()` call always returns a tag that this process owns. The algorithm ensures that when a process returns the first free tag of some block b_i in a `GetFree()` operation, then not only is that tag free, but all tags in that block are. During its next $\delta - 1$ `GetFree()` operation calls the process returns only tags from block b_i, and with every such operation call it executes a constant amount of work to move toward identifying a new block that contains only free tags. We now explain how a process finds such a block.

To enable faster identification of free tags from (potentially) occupied ones, the second idea is that instead of checking each tag individually, we aggregate each process' information of a whole block into two numbers. For this purpose, two ABA-detecting register arrays, each of size βn, are utilized for each process p: Act_p is MWMR, and Emp_p is SWMR. Process p increments the value of $Act_p[i]$ during its `GetFree()` call that returns tag $g \in b_i$; in a `Release(`g`)` operation call, it decrements $Act_p[i]$. Therefore, the algorithm maintains the invariant that $\sum_{p=0}^{n-1} Act_p[i]$ is the total number of tags in block b_i that are active (except while registers $Act_0[i],\ldots,Act_{n-1}[i]$ are being reset, as later descried).

The value of $Emp_p[i]$ gets incremented in a `Protect(`g`)` call by p and gets decremented in `Unprotect(`g`)` and `CancelProtect(`g`)` calls by p. We say a tag g is **employed** k times by process p at some point t if that process called `Unprotect(`g`)` and `CancelProtect()` together ℓ times for some value $\ell \geq 0$, and `Protect(`g`)` $k+\ell$ times before t. Operations `Protect()` and `Unprotect()` are not only used as external operations, but as sub-routines for all other

operations. These operations are called in such a way that it is ensured that

$$\text{if } p \text{ protects tag } g\text{, then } g \text{ is employed at least once by } p\text{, and} \tag{7.3}$$

$$\text{if } g \text{ is stored in } A\text{, then } g \text{ is employed at least once by some process.} \tag{7.4}$$

However, a tag g that is employed at least once is not necessarily protected or stored in A, but a process might have decided to conservatively employ tag g, until the process can certainly decide about the state of g.

Let $E(i) = \sum_{p=0}^{n-1}(Emp_p[i] + Act_p[i])$. The value of $E(i)$ is at least (because tags can be employed multiple times) the total number of tags in block b_i that are employed or active. So by (7.3) and (7.4), if at least one tag in block b_i is occupied, then $E(i) > 0$. Thus, our algorithm maintains the invariant that

$$\text{if } E(i) = \sum_{p=0}^{n-1}(Emp_p[i] + Act_p[i]) = 0\text{, then all tags in block } b_i \text{ are free.} \tag{7.5}$$

In a `GetFree()` operation, in order to find a block that contains only free tags, it suffices for process p to find an index i such that it owns b_i and $E(i) = 0$. Process p checks if $E(i) = 0$ using $O(n)$ steps as follows: It reads the ABA-detecting registers $Emp_0[i],\ldots,Emp_{n-1}[i]$ and $Act_0[i],\ldots,Act_{n-1}[i]$, and computes the sum of their values in sum_p. Then, p reads these ABA-detecting registers again, and uses the boolean values returned by `DRead()` operations to ensure that none of the register values has changed; essentially this is a double collect as in the standard snapshot implementation (Afek, Attiya, Dolev, Gafni, Merritt and Shavit, 1993). If $sum_p = 0$ and no register had changed in the second collect, then there was a point at which $E(i) = 0$, and thus at that point all tags in b_i were free.

Our implementation guarantees that among the β blocks owned by process p, there is always at least one block b_i, such that $E(i) = 0$ throughout the interval during which p is looking for a block with free tags. As a result it takes $O(n\beta)$ steps to find a free block. We distribute this work over $O(n\beta)$ of p's `GetFree()` operations, during each of which a constant number of the

required steps get executed. A sufficiently large block size of $\delta = 2\beta n + n$ guarantees that p finds a new block with $E(i) = 0$ before it runs out of tags in its current block.

So far, each process p only needs to increment or decrement "its own" ABA-detecting registers $Emp_p[i]$ or $Act_p[i]$. Therefore, reading the current value and incrementing it is done without any interruption from other processes.

A problem is that even when $E(i) = 0$, each individual register $Act_q[i]$ can be positive or negative, even though we always have $\sum_{q=0}^{n-1} Act_q[i] \geq 0$: Process q may have decreased the value of $Act_q[i]$ when releasing a tag from b_i, while a different process p originally obtained that tag from a `GetFree()` operation during which it incremented $Act_p[i]$. To stop the values of those registers from growing very large over time, p resets all registers $Act_q[i]$, $q \in \{0,\ldots,n-1\}$, to 0, once it determines $E[i] = 0$, by executing n `DWrite()` operations throughout n `GetFree()` operation calls. The fact that all tags of block b_i are free guarantees that no other process concurrently accesses any of those registers, while they are being reset.

7.3.2 Reading and Writing in TRA

A register A is used to store the value of the object. To execute a `TRead()` operation R on the TRA object, a process must perform two tasks:

(I) It has to identify a tag that is stored in A at some point t during R (t will be the linearization point of R), and

(II) it has to ensure that this tag is employed from point t onwards.

The second task is required, because of (7.3), and the fact that this process protects g one additional time starting from the linearization point of R.

To achieve these, the exact same hint mechanism of Chapter 4 and its deamortization technique are used. What we get is the following. There are two hint arrays, and each process p maintains two counters, $hCtr_p$ and $ReadCtr_p$. The value of $hCtr_p$ indicates to which process p is going to provide hint next, and p increments it modulo n with each `TWrite()` call. The

value of $ReadCtr_p$ is incremented by one during each `TRead()` call by p. (Suppose for now that this counter is unbounded. The same approach as in Chapter 4 is used to bound this counter.) During p's `TWrite()` call, p ensures that the process with id $q = hCtr_p$, has a hint from p that is augmented with the current value of $ReadCtr_q$.

Since p deamortizes the work of providing hints to all other processes into n `TWrite()` calls, it has to ensure that any tag that p writes to A during its last n `TWrite()` calls, as well as any tags that is stored in hint arrays and is written by p are employed. For that, p maintains a local queue which keeps track of all those tags, and it ensures that each tag in the queue is employed.

Suppose q reads tag g from A during R, and p is the process which wrote this value to A. The hint mechanism ensures that

(a) either p has provided a tag g' as a hint to q (and so $A = g'$) at some point during R, and p has g' employed at least until q invokes another `TRead()`, or

(b) p has tag g employed until q reads the hint array.

To identify which of (a) and (b) hold during R, process q reads the hints provided from p. If the hints are the same and are augmented with the current value of $ReadCtr_q$, then (a) holds. So q returns g', and by (a), $A = g'$ at some point t during this call, and t is the linearization point of R. Otherwise (b) holds, and q returns tag g that it reads from A. This operation lienarizes at the read of A. Thus, (I) is guaranteed.

Now we explain how q guarantees (II). Process q calls `Protect(`g`)` right after its read of g from A. If it decides to return the hint tag g', then it calls `Protect(`g'`)` and `Unprotect(`g`)`. This way, by (a) and (b) the return value of R is employed at first by p, and then by q starting from the point at which $A = g'$, which is the linearization point of R.

The pseudocode for `TRead()` and `TWrite()` is provided in Figure 7.7.

```
// Tag domain T = {0,...,nβδ − 1}
// β = mn(2n + 5) + τ + 3n + 1
// δ = 2nβ + n

shared:
  ∀p ∈ [n]: ABA-detecting register Emp_p[nβ]
  ∀p ∈ [n]: ABA-detecting register Act_p[nβ] = 0

local to process p, and with global scope:
  int tag_p = pβ − 1
  int ρ_p = 0, j_p = 0
  int sum_p = 0
  boolean sum'_p = false
  int emp_p[nβ] = {0,...,0}
```

Operation $\texttt{GetFree}_p()$

```
61 tag_p++
62 (x, f) := Emp_{ρ_p mod n}[pβ + j_p].DRead()
63 (x', f') := Act_{ρ_p mod n}[pβ + j_p].DRead()
64 if ρ_p < n then sum_p := sum_p + x + x'
65 if n ≤ ρ_p < 2n then sum'_p := sum'_p ∨ f ∨ f'
66 (ρ_p++) mod 3n
67 if ρ_p = 2n ∧ (sum_p ≠ 0 ∨ sum'_p = true) then
68     sum_p := 0; sum'_p := false; ρ_p := 0
69     (j_p++) mod β
70 else if 2n ≤ ρ_p < 3n then
71     Act_{ρ_p mod n}[pβ + j_p].DWrite(0)
72 else if ρ_p = 0 then
73     tag_p := (pβ + j_p) × δ
74     (j_p++) mod β
75 u := Act_p[⌊tag_p/δ⌋].DRead()
76 Act_p[⌊tag_p/δ⌋].DWrite(u + 1)
77 return tag_p
```

Operation $\texttt{Protect}_p(g)$

```
78 emp_p[⌊g/δ⌋]++
79 Emp_p[⌊g/δ⌋].DWrite(emp_p[⌊g/δ⌋])
```

Operation $\texttt{Unprotect}_p(g)$

```
80 emp_p[⌊g/δ⌋]--
81 Emp_p[⌊g/δ⌋].DWrite(emp_p[⌊g/δ⌋])
```

Operation $\texttt{CancelProtect}_p(g)$

```
82 emp_p[⌊g/δ⌋]--
83 Emp_p[⌊g/δ⌋].DWrite(emp_p[⌊g/δ⌋])
```

Operation $\texttt{Release}_p(g)$

```
84 u := Act_p[⌊g/δ⌋].DRead()
85 Act_p[⌊g/δ⌋].DWrite(u − 1)
```

Figure 7.6: Managing Tags in (m,τ)-TRA and (m,τ)-TLSA Object

shared:
  register $A[m]$
  register $H[m][n][n]$
  register $H'[m][n][n]$
  $\forall p \in [n]$: **register** $ReadCtr_p[m]$

local to process p, and with global scope:
  boolean $toggle_p[m][n] = 0$
  int $hCtr_p[m] = 0$
  Queue $rsrvQ_p[m]$ (initially contains 2n+4 elements of value $\perp$)

Operation $\texttt{TWrite}_p(i,(x,g))$

```
q := hCtr_p[i]
c := ReadCtr_q[i].Read()
Protect(g)
(x_1,g_1,c_1,b_1) := H[i][p][q].Read()
(x_2,g_2,c_2,b_2) := H'[i][p][q].Read()
updateQ(i,g)
b := toggle_p[i][q]
if c ≠ c_1 ∨ (x_1,g_1,c_1,b_1) ≠ (x_2,g_2,c_2,b_2) then
    H[i][p][q].Write(x,g,c,b)
    c' := ReadCtr_q[i].Read()
    A[i].Write(x,g,p)
    H'[i][p][q].Write(x,g,c',b)
else
    Protect(g_1)
    updateQ(i,g_1)
    A[i].Write(x,g,p)
toggle_p[i][q] := 1 − b
(hCtr_p[i]++) mod n
```

Operation $\texttt{updateQ}_p(i,g)$

```
rsrvQ_p[i].enq(g)
g' := rsrvQ_p[i].deq()
if g' ≠ ⊥ then Unprotect(g')
```

Operation $\texttt{TRead}_p(i)$

```
c := (ReadCtr_p[i].Read() + 1) mod 2n
H[i][c mod n][p].Write(⊥,⊥,⊥,⊥)
H'[i][c mod n][p].Write(⊥,⊥,⊥,⊥)
ReadCtr_p[i].Write(c)
(x,g,q) := A[i].Read()
retv := (x,g)
Protect(g)
(x_1,g_1,c_1,b_1) := H[i][q][p].Read()
(x_2,g_2,c_2,b_2) := H'[i][q][p].Read()
if c = c_1 ∧ (x_1,g_1,c_1,b_1) = (x_2,g_2,c_2,b_2) then
    retv := (x_1,g_1)
    Unprotect(g)
    Protect(g_1)
return retv
```

Figure 7.7: TWrite() and TRead() of an (m,τ)-TRA Object

shared:
LL/SC $A[m]$
LL/SC $H[m][n]$

local to process p, and with global scope:
boolean $flag_p[m] = 0$
int $hCtr_p[m] = 0$
queue $rsrvQ_p[m]$ (initially contains 2n+4 elements of value $\perp$)

Operation $\texttt{TSC}_p(i,(x,g))$

```
q := hCtr_p[i]
if flag_p[i] = 1 then return false
Protect(g)
(x',g',p') := H[i][q].LL()
if A[i].SC(x,g) = false then
    Unprotect(g)
    return false
updateQ(i,g)
if (x',g',p') = (⊥,⊥,⊥) then
    H[i][q].SC(x,g,p)
else if p' = p then
    Protect(g')
    updateQ(i,g')
(hCtr_p[i]++) mod n
return true
```

Operation $\texttt{updateQ}_p(i,g)$

```
rsrvQ_p[i].enq(g)
g' := rsrvQ_p[i].deq()
if g' ≠ ⊥ then Unprotect(g')
```

Operation $\texttt{TLL}_p(i)$

```
flag_p[i] = 0
repeat
    H[i][p].LL()
until H[i][p].SC(⊥,⊥,⊥)
retv := (x,g) := A[i].LL()
Protect(g)
(x',g',p') := H[i][p].LL()
if (x',g',p') ≠ (⊥,⊥,⊥) then
    retv := (x',g')
    Unprotect(g)
    (x'',g'') := A[i].LL()
    if (x'',g'') ≠ (x',g') then
        flag_p[i] := 1
    Protect(g')
return retv
```

Figure 7.8: Load and Store of an (m,τ)-TLSA Object

7.3.3 Loading and Storing in TLSA

In this implementation, A is an LL/SC object. Similar to TRA, to execute a `TLL()` operation R on the TLSA object, a process p must accomplish both (I) and (II). In addition, it must be ensured that

(III) q holds a valid link to the TLSA object if and only if no `TSC()` call has linearized since the linearization point of q's last `TLL()` call.

To achieve these, we again employ the idea of hint mechanism of Chapter 4. However, the availability of LL/SC objects allows a simpler implementation of the hint technique: Instead of using $2n$ registers for the hints that other processes provide to each process q, one LL/SC object is used. Process q resets this register to $\perp$ at the beginning of each `TLL()` call. So if q later reads a non-$\perp$ value from this hint entry, then this hint is surely provided at some point during q's ongoing `TLL()` call. During a `TSC(`g`)` call by p, it only provides g as a hint to the process with id $q = hCtr_p$, if p first loads $\perp$ from the hint register for q, and then it successfully changes the value of A via an A.`SC(`g`)` call.

The rest of the hint mechanism is the same as the one from TRA implementation. Therefore, if q reads tag g from A during R, then there is some process p, such that (a) and (b) are true for p.

Now consider a `TLL()` call R by q in which q loads some value g from its hint entry. R returns g if $g \neq \perp$, and otherwise it returns the value it loads from A. To show (I)-(III), we first define the linearization point of each operation. An unsuccessful `TSC()` call linearizes at its response. A successful `TSC()` linearizes at its successful A.`SC()` call. The linearization point of a `TLL()` call R by p that returns the value that p loads from A is that A.`LL()` call, so (I) is true for this case. Moreover in this case, p has a valid link to TLSA if and only if p holds a valid link to A. Now suppose R returns the hint value g. Process p executes a second A.`LL()` call after it decides that it is going to return g. If the value it loads from A is g again, then R linearizes at this point, so (I) follows. However, if it does not read g from A, p knows that $A = g$ at an earlier

point t (by (a)), but now is not. So a successful `TSC()` must have linearized since t. Process p remembers this by setting a boolean register $flag_p$, that was reset at the beginning of its `TLL()` call. Operation R then linearizes at t, and so (I) is maintained. As a result, we conclude that no matter where R linearizes,

$$\begin{aligned} &p \text{ has a valid link to the TLSA object if and only if} \\ &p \text{ holds a valid link to } A \text{ and } flag_p \neq 1. \end{aligned} \tag{7.6}$$

Thus, to guarantee (III), it is enough that during a `TSC()` call, p first reads $flag_p$, and returns false immediately if $flag_p = 1$, and only otherwise it executes an A.`SC()`. Finally, (a) and (b) ensure that some process p has the return value of R employed at least up to the point that q employs this value during R, and so (II) follows.

7.4 Detailed Description of TRA and TLSA

In this section, we provide a more detailed description of the implementation of TRA and TLSA provided in Figures 7.6–7.8. Recall that for ease of explanation, we describe the code for $(1,\tau)$-TRA and $(1,\tau)$-TLSA objects, and we use X, instead of $X[0]$, for any array X that its first dimension has m elements. The pseudocode and proofs, however, are for the general taggable arrays of size m.

7.4.1 Managing Tags

The domain of tags $T = \{0,\ldots,\Delta(m,n,\tau)-1\}$ is partitioned into $\beta \cdot n$ blocks $b_0,\ldots,b_{\beta n-1}$, and $\beta = mn(2n+5)+\tau+3n+1$. Each block contains $\delta = 2\beta n + n$ tags. Block b_i contains tags $i\cdot\delta,\ldots,i\cdot\delta+\delta-1$ and is owned by process $\lfloor i/\beta \rfloor$.

The GetFree() Operation. During this operation, process p has to perform two main tasks: progress with finding a block with free tags, and prepare a tag to be returned. A local variable tag_p keeps track of the tag that p's last `GetFree()` operation returned. Let $c = \lfloor tag_p/\delta \rfloor$ be

the index of the block to which tag_p belongs. In line 61, process p increments tag_p. For the reasons that we discuss later, p always switches to a new block before it runs out of tags in its current block, b_c. So after line 61, the value of tag_p is the next free tag in block b_c. Before this process returns this tag, in lines 75 and 76, p increments $Act_p[c]$ to indicate that the tag it returns is now active. This is done by executing a `DRead()` followed by a `DWrite()`. These two steps are executed without an overlapping modification of $Act_p[c]$, because process p only returns tags that it owns, so p owns block b_c, and so only p writes to $Act_p[c]$. The `GetFree()` operation linearizes with the write to $Act_p[c]$.

Recall that our algorithm maintains the invariant that if $E(i) = \sum_{p=0}^{n-1}(Emp_p[i] + Act_p[i]) = 0$, then all tags in block b_i are free (see Claim 7.23). The rest of the operation is devoted to performing a constant number of steps of the work required for finding a new free block, that is, a block b_i for which $E(i) = 0$. Variable j_p keeps track of the block that p is currently checking whether it is free. More precisely, the block being checked has index $i = p\beta + j_p$. Variable $\rho_p \bmod n$ indicates the process whose variables $Emp_{\rho_p \bmod n}[i]$ and $Act_{\rho_p \bmod n}[i]$ are being read and summed up in this `GetFree()` call. Over time, each of those variables needs to be read twice to perform a double collect on all these registers, and, if b_i is identified as a free block, then the variables $Act_0[i],\ldots,Act_{n-1}[i]$ need to be reset. Therefore, ρ_p takes values in $\{0,\ldots,3n-1\}$.

In lines 62–63, p uses `DRead()` to read $Emp_{\rho_p \bmod n}[p\beta + j_p]$ and $Act_{\rho_p \bmod n}[p\beta + j_p]$. If $\rho_p < n$, then this is the first time p reads those variables during the current double collect, so it sums up the returned values in line 64 into a local variable sum_p. If $n \leq \rho_p < 2n$, then this is the second time that p is reading those variable during the current double collect, so it adds the flags returned from the `DRead` operations to its local variable sum'_p in line 65. Recall that each flag is `false` if the corresponding value has not changed since the previous read, and otherwise it is `true`. After that p increments ρ_p modulo $3n$ in line 66.

If $\rho_p = 2n$ after the increment, then p has completed its double collect. So if $sum_p \neq 0$ or $sum'_p \neq$ `false`, then p has witnessed that $E(i) > 0$ at some point (see Claim 7.27(b)), and so

the process prepares to move on to check a new block in its subsequent `GetFree()` calls. (It is possible for p to determine that $E(i) \neq 0$ before $\rho_p = 2n$, but in this implementation, for the sake of simplicity, we did not try to make this part efficient.) To that end, p increments $j_p \bmod \beta$, and resets ρ_p, sum_p, and sum'_p in lines 68–69. Otherwise, if $sum_p = 0$ and $sum'_p = \texttt{true}$, then p has witnessed that $E(i) = 0$ at some point during its double collect (see Claim 7.27(a)). Hence the process identified a block b_i whose all tags are free. The algorithm ensures once all tags of a block are free, they all remain free until the first tag of this block is returned from a `GetFree()` operation (see Claim 7.24). In this case, in line 71 of this and the next $n-1$ `GetFree()` calls, while $2n \leq \rho_p < 3n$, p resets $Act_0[i],\ldots,Act_{n-1}[i]$. If $\rho_p = 0$, then p has made $3n$ `GetFree()` calls that dealt with block b_i, for $i = p\beta + j_p$, and thus it identified b_i as free and reset all processes' Act variables of that block. Hence, the `GetFree()` operation returns the first tag, $(p\beta + j_p) \cdot \delta$, from block b_i. Therefore, p sets tag_p to that value in line 73. In line 74 process p then increments j_p modulo β so that it begins its search for another free block in its next `GetFree()` call. Finally in line 77, process p returns tag tag_p (after updating the corresponding entry of Act_p).

Protect(), Unprotect(), CancelProtect(), and Release(). For a tag $g \in b_i$, operation `Protect(`g`)` by p only increments the value of SWMR ABA-detecting register $Emp_p[i]$ by 1, while `Unprotect(`g`)` and `CancelProtect(`g`)` by p decrement the value stored in this register by 1. In operation `Release(`g`)`, process p decrements $Act_p[i]$, using a `DRead()` followed by a `DWrite()`. Only p and the owner of block b_i can write to $Act_p[i]$. The algorithm ensures that the owner does not do so, while p executes `Release(`g`)` (see Claim 7.32).

7.4.2 Reading and Writing in TRA

Register A stores the value of the TRA object, and two register arrays H and H' are used, where $H[p][q]$ and $H'[p][q]$ represent the hint process p provides to process q, for $p,q \in \{0,\ldots,n-1\}$. Only processes p and q read from and write to $H[p][q]$ and $H'[p][q]$.

The TWrite() Operation. Consider a `TWrite(`g`)` operation W by p. To guarantee Property (b) (from page 171), process p has to ensure that once it writes a tag g into A, this tag remains employed for p's next n `TWrite()` calls, so that p either provides a hint or finds an up-to-date hint for each process. For that, process p enqueues tag g into a local queue, called $rsrvQ_p$, by calling `updateQ(`g`)`. This tag remains in this queue for at least the next n `TWrite()` calls by p. The algorithm ensures that any tag stored in this queue is employed, by calling `Protect(`g`)` before enqueuing tag g, and calling `Unprotect(`g'`)` after dequeuing tag g' from this queue. So p executes `Protect(`g`)` and `updateQ(`g`)` in lines 88 and 91 as this process is about to write g into A.

In each `TWrite()` call, process p chooses a different process (in a round robin fashion) whose id is stored in p's local variable $hCtr_p$, and tries to provide a hint to that process. Let q be the value stored in $hCtr_p$ at the beginning of W (line 86). Process p, in line 87, reads the current value c of $ReadCtr_q$, which is the counter that process q increments modulo $2n$ once in every `TRead()` call q executes. Then p reads $H[p][q]$ and $H'[p][q]$ in lines 89 and 90. Process only provides a hint to q, if the values it reads from $H[p][q]$ and $H'[p][q]$ do not match, or the counter value augmented to those hints is not c (line 93). In this case, in lines 94–97, just before it writes to A, the writer p provides g as a hint to q by writing g, c, as well as the current value b of its local register $toggle[q]$ to $H[p][q]$. $toggle[q]$ is a local register that p toggles during each `TWrite()` call in which $hCtr_p = q$, and is used to help process q to distinguish hints provided from q in different `TWrite()` calls. (In Chapter 4, we use queues that are twice as large as used in place of using this register.) Then p writes g and its own id to A, reads $ReadCtr_q$ again into c', and writes (g,c',b) into $H'[p][q]$. Recall from Chapter 4, that we use the second hint array to avoid using unbounded values for $ReadCtr_q$.

Now suppose that p reads (g_1,c,b_1) from both $H[p][q]$ and $H'[p][q]$, for some g_1 and b_1. This means there is already an up-to-date hint for q from p. Process p has to ensure that g_1 remains employed at least until $hCtr_p = q$ again, and so during p's next n `TWrite()` calls. For

that reason, p employs g_1 by calling `Protect(`g_1`)`, and enqueues g_1 into $rsrvQ_p$ by calling `updateQ(`g_1`)` (lines 99 and 100). Then p writes g and its own id to A in line 101. Finally, p toggles the bit stored in $toggle[q]$ and increments its counter $hCtr_p$ modulo n (lines 102 and 103). This operation always linearizes at the point p writes g into A.

The TRead() Operation. During a `TRead()` operation by some process p, the process increments its counter $ReadCtr_p$ modulo $2n$. As $ReadCtr_p$ is a single writer register, in order to increment this counter, process p reads some value c' from its $ReadCtr_p$ in line 107, and writes $c = (c' + 1) \bmod 2n$ into the same register in line 110. But just before it updates the value of $ReadCtr_p$ to c, process p resets the value stored in $H[c \bmod n][p]$ and $H'[c \bmod n][p]$ by writing $(\perp, \perp, \perp)$ into those registers in lines 108 and 109. This is to ensure that each hint entry of $H[0 \ldots n-1][p]$ gets reset once in every n `TRead()` calls by p, which prevents p from using an outdated hint.

Then in line 111, this process reads some tag g from A and the id q of the process which wrote this value. Process p employs g by calling `Protect(`g`)`, and reads both $H[q][p]$ and $H'[q][p]$ (in lines 113–115). If p reads different values from these two registers, or the counter value augmented to one of them is not c, then p knows that q has not executed n `TWrite()` calls after it wrote g to A, as otherwise, q would have made sure that p would have an up-to-date hint (see Claim 7.15). So tag g must have been continuously employed from the point $A = g$ and until p read the hint registers. Thus, p returns tag g in this case. This operation linearizes at the point p reads g from A.

If both $H[q][p]$ and $H'[q][p]$ contain (g_1, c, b_1), for some g_1 and b_1, then p has a hint that is provided from q since p incremented its $ReadCtr_p$ in line 110 (see Claim 7.14). Therefore in lines 117–120, p chooses g_1 as its return value, and calls `Protect(`g_1`)`. Before p returns g_1, it calls `Unprotect(`g`)`, because g is not the return value of this operation anymore in this case. This operation linearizes at the point at which q wrote g_1 to A, which is guaranteed to be at some point during p's ongoing `TRead()` call.

7.4.3 Loading and Storing in TLSA

In this implementation, A is a LL/SC object that stores the value of the TLSA object. A hint is provided to process q in $H[q]$, where H is an array of LL/SC objects of size n.

The TSC() Operation. At the beginning of a `TSC(`g`)` operation by p on the TLSA object, process p first checks its local variable $flag_p$, and if that is set, it returns `false` immediately (line 122). This is because p in its last `TLL()` call has witnessed a change of the value stored in A since the linearization point of that operation. Thus, p does not hold a valid link to the object. Otherwise, since p is about to make an attempt to write g into A, it employs g by calling `Protect(`g`)` in line 123. Then, in line 124, process p loads $H[q]$, where q is the value stored in p's local register $hCtr_p$ and indicates the process to which p may provide a hint in this `TSC()` operation. ($hCtr_p$ changes in a round robin fashion with each of p's `TSC()` calls in line 134.) Then an attempt is made to store g into A using an A.`SC(`g`)` operation (line 125). If that attempt fails, then p unemploys g again by calling `Unprotect(`g`)` and returns `false` to indicate that its `TSC()` on the TLSA object failed (lines 126–127). If A.`SC(`g`)` call succeeds, then the point when this successfully happens will be the linearization point of p's `TSC()` operation on the TLSA object. Then p calls `updateQ(`g`)` in line 128. During this operation, p enqueues g into its local queue $rsrvQ$. The implementations of `TSC()` and `updateQ()` ensure that after this point, g remains in the queue, and so employed, throughout p's next $n+1$ successful `TSC()` operations (see Claim 7.9).

Then p makes an attempt to provide g as a hint to process q, if $H[q]$ does not already contain a hint. That is, g gets stored into $H[q]$ if and only if $H[q] = (\perp, \perp)$ when p loaded it earlier, and $H[q]$ has not changed since then (lines 129–130). If $H[q]$ already contains a hint, g', previously provided by p, then p calls `Protect(`g'`)` followed by `updateQ(`g'`)` to ensure that the hint remains employed for another n successful `TSC()` operations by p (lines 131–133). (If the current hint was provided by $p' \neq p$, then p' ensures that this tag remains protected.) This is to guarantee that any hint provided by p remains employed as long as it is stored in H. Finally, p

increments $hCtr_p$ modulo n and returns `true` (lines 134–135).

The TLL() Operation. During a `TLL()` operation L on the TLSA object, process p first resets $flag_p$ in line 139. This is because p has to ensure that $flag_p$ is only 1 if p does not succeed to make a valid link to object A at the linearization point of this operation. In lines 140–142, process p resets $H[p]$ by repeating $H[p]$.`LL()` and $H[p]$.`SC(`$\perp,\perp$`)` operations, until an $H[p]$.`SC(`$\perp,\perp$`)` succeeds. Since any other process will only change the value of $H[p]$ when its value is $(\perp,\perp)$, p needs at most two attempts to reset $H[p]$ (see the proof of Claim 7.14). Then p executes an A.`LL()` operation to obtain a tag g, and since g is tentatively the return value of this operation, p employs g by calling `Protect(`g`)` (lines 143–144). After that p reads $H[p]$ in line 145. If $H[p] = (\perp,\perp)$, then no process provided a hint to p since p reset $H[p]$ at the beginning of L. In particular, this is true for the process q that stored tag g in A prior to p's A.`LL()`. This implies that q has executed at most $n-1$ complete and successful `TSC()` operation calls since then, because in n consecutive successful `TSC()` operation calls, q attempts to provide a hint to every process (see Claim 7.15(a)). Since tag g remains employed throughout n successful `TSC()` operation calls by q, following the one in which q stored it into A, tag g is still employed by q when p reads $(\perp,\perp)$ from $H[p]$; specifically, it has been continuously employed starting from the point when p read g from A until p employed g itself prior to reading $H[p]$ (see Claim 7.15(b)-(c)).As a result, p can return g from its `TLL()` operation call in line 153. In this case, the linearization point of L is when q loads g from A in line 143.

Now suppose that $H[p]$ contains a pair $(g',p') \neq (\perp,\perp)$ when p loads it in line 145. Then it is ensured that, during a successful `TSC(`g'`)` operation S', process p' must have executed an $H[p]$.`LL()` and a subsequent successful $H[p]$.`SC(`g',p'`)` operation that stored g' into A (in line 125). This must have happened after p reset $H[p]$ in line 142, and before it loaded (g',p') from $H[p]$ in line 145 of L (see Claim 7.14(a)-(b)). Thus, L can return tag g'. Moreover, process p' ensures that g' remains employed as long as the pair (g',p') remains in $H[p]$, and only p can reset that LL/SC object. Hence, p can now simply call `Protect(`g'`)` in line 152 and be sure that

g' has continuously been employed since the point p' wrote g' into A. Since L will not return tag g, p also calls `Unprotect(g)` in line 148.

Now p has to decide the value of $flag_p$, and based on that, the linearization point of R is determined. Process p knows that at the linearization point of S', $A = g'$, and g' has continuously been employed since then. But p also has to ensure that either p holds a valid link to A starting from a point that $A = g'$, or $flag_p$ must be set to `false`. For that, process p loads A one more time after it has decided to use hint g' (line 149). If the value of A is equal to g', then p does not set $flag_p$, and L will linearize at the point of p's last load of A (line 149), when the value is g'. If the value of A is not equal to g', then p sets the local flag $flag_p$ (lines 150–151), and linearizes at the point at which p' wrote g' to A at the linearization point of S'. The fact that A does not contain g' implies that a successful `TSC()` must have linearized since the linearization point of L. This will force p's next `TSC()` operation to fail immediately (line 122).

7.5 Correctness of TLSA and TRA

In the following section, we give the correctness proof of the TLSA and TRA implementations. As both proofs are similar with only minor differences, we combine the correctness proofs for both implementations as much as possible. To that end, unless it is clearly mentioned, each statement applies to both implementations. However, if the two correctness proofs require different arguments, then we either clearly mention which implementation we are considering, or we use the following notation: To differentiate between arguments for the TLSA vs TRA implementation, we distinguish between them using different colors and brackets. Blue text in double square brackets, $[\![\,]\!]$, refers only to the TLSA implementation, and plum colored text in angled brackets, $\langle\rangle$, refers only to the TRA implementation. For example, statement "if H is a history on TLSA, then H satisfies property X, and if H is a history on TRA, then it satisfies property Y" could be expressed as "If H is a history on $[\![\text{TLSA}]\!]\langle\text{TRA}\rangle$, then H satisfies property $[\![\text{X}]\!]\langle\text{Y}\rangle$".

shared:
  LL/SC $A[m]$
  LL/SC $H[m][n]$

local to process p, and with global scope:
  boolean $flag_p[m] = 0$
  int $hCtr_p[m] = 0$
  queue $rsrvQ_p[m]$ (initially contains 2n+4 elements of value $\perp$)

Operation $\texttt{TSC}_p(i,(x,g))$

```
154 q := hCtr_p[i]
155 if flag_p[i] = 1 then return false
156 reserve(g)
157 (x',g',p') := H[i][q].LL()
158 if A[i].SC(x,g) = false then
159     unreserve(g)
160     return false
161 updateQ(i,g)
162 if (x',g',p') = (⊥,⊥,⊥) then
163     H[i][q].SC(x,g,p)
164 else if p' = p then
165     reserve(g')
166     updateQ(i,g')
167 (hCtr_p[i]++) mod n
168 return true
```

Operation $\texttt{TLL}_p(i)$

```
169 flag_p[i] = 0
170 repeat
171     H[i][p].LL()
172 until H[i][p].SC(⊥,⊥,⊥)
173 retv := (x,g) := A[i].LL()
174 announce(g)
175 (x',g',p') := H[i][p].LL()
176 if (x',g',p') ≠ (⊥,⊥,⊥) then
177     retv := (x',g')
178     unannounce(g)
179     (x'',g'') := A[i].LL()
180     if (x'',g'') ≠ (x',g') then
181         flag_p[i] := 1
182     announce(g')
183 return retv
```

Figure 7.9: Revised Implementation of an (m,n,τ)-TLSA Object

7.5.1 Modified Pseudocode of the TLSA and TRA Implementations

In order to accommodate the proofs, we change the pseudocode of Figures 7.6–7.8 to the one in Figures 7.9–7.11. The modifications, as explained in the following, do not change the behaviour of the algorithm. First, to distinguish whether an entry of *Emp* is modified in a $[\![\texttt{TLL()}]\!]\langle\texttt{TRead()}\rangle$ or in other operations, we introduce four new operations, `reserve(g)`, `unreserve(g)`, `announce(g)`, and `unannounce(g)`. New operations `reserve(g)` and `announce(g)` are the same as `Protect(g)` in the original algorithm, and operations `unreserve(g)` and `unannounce(g)` are the same as `Unprotect(g)` of the original algorithm. In the modified pseudocode, operations `Protect(g)` and `Unprotect(g)` simply call `announce(g)` respectively `unannounce(g)`. Moreover, we let operation $[\![\texttt{TLL()}]\!]\langle\texttt{TRead()}\rangle$ call `announce(g)` and `unannounce(g)`, and $[\![\texttt{TSC()}]\!]\langle\texttt{TWrite()}\rangle$ (and therefore `updateQ()`) call

// Tag domain $T = \{0,\ldots,n\beta\delta - 1\}$
// $\beta = mn(2n+5) + \tau + 3n + 1$
// $\delta = 2n\beta + n$

shared:
$\forall p \in [n]$: **ABA-detecting register** $Emp_p[n\beta]$
$\forall p \in [n]$: **ABA-detecting register** $Act_p[n\beta] = 0$

Operation $\texttt{GetFree}_p()$

```
tag_p++
(x, f) := Emp_{ρ_p mod n}[pβ + j_p].DRead()
(x', f') := Act_{ρ_p mod n}[pβ + j_p].DRead()
if ρ_p < n then sum_p := sum_p + x + x'
if n ≤ ρ_p < 2n then sum'_p := sum'_p ∨ f ∨ f'
(ρ_p++) mod 3n
if ρ_p = 2n ∧ (sum_p ≠ 0 ∨ sum'_p = true) then
    sum_p := 0; sum'_p := false; ρ_p := 0
    (j_p++) mod β
else if 2n ≤ ρ_p < 3n then
    // σ(p, pβ + j_p) += Act_{ρ_p mod n}[pβ + j_p]
    Act_{ρ_p mod n}[pβ + j_p].DWrite(0)
else if ρ_p = 0 then
    tag_p := (pβ + j_p) × δ
    (j_p++) mod β
u := Act_p[⌊tag_p/δ⌋].DRead()
Act_p[⌊tag_p/δ⌋].DWrite(u + 1)
return tag_p
```

Operation $\texttt{Release}_p(g)$

```
u := Act_p[⌊g/δ⌋].DRead()
Act_p[⌊g/δ⌋].DWrite(u − 1)
```

Operation $\texttt{updateQ}_p(i,g)$

```
rsrvQ_p[i].enq(g)
g' := rsrvQ_p[i].deq()
if g' ≠ ⊥ then unreserve(g')
```

local to process p, and with global scope:
int $tag_p = p\beta - 1$
int $\rho_p = 0, j_p = 0$
int $sum_p = 0$
boolean $sum'_p = \texttt{false}$
int $emp_p[n\beta] = \{0,\ldots,0\}$

Operation $\texttt{Protect}_p(g)$

```
announce(g)
```

Operation $\texttt{Unprotect}_p(g)$

```
unannounce(g)
```

Operation $\texttt{CancelProtect}_p(g)$

```
unannounce(g)
```

Operation $\texttt{reserve}_p(g)$

```
emp_p[⌊g/δ⌋]++
Emp_p[⌊g/δ⌋].DWrite(emp_p[⌊g/δ⌋])
```

Operation $\texttt{unreserve}_p(g)$

```
emp_p[⌊g/δ⌋]--
Emp_p[⌊g/δ⌋].DWrite(emp_p[⌊g/δ⌋])
```

Operation $\texttt{announce}_p(g)$

```
emp_p[⌊g/δ⌋]++
Emp_p[⌊g/δ⌋].DWrite(emp_p[⌊g/δ⌋])
```

Operation $\texttt{unannounce}_p(g)$

```
emp_p[⌊g/δ⌋]--
Emp_p[⌊g/δ⌋].DWrite(emp_p[⌊g/δ⌋])
```

Figure 7.10: Additional Operations for Revised Implementation of an (m,τ)-TLSA and (m,n,τ)-TRA

```
shared
  register A[m]
  register H[m][n][n]
  register H'[m][n][n]
  ∀p ∈ [n]: register ReadCtr_p[m]

local to process p, and with global scope:
  boolean toggle[m][n] = 0
  int hCtr_p[m] = 0
  queue rsrvQ_p[m] (initially contains 2n+4 elements of value ⊥)

Operation TWrite_p(i,(x,g))
217 q := hCtr_p[i]
218 c := ReadCtr_q[i].Read()
219 reserve(g)
220 (x1,g1,c1,b1) := H[i][p][q].Read()
221 (x2,g2,c2,b2) := H'[i][p][q].Read()
222 updateQ(i,g)
223 b := toggle_p[i][q]
224 if c ≠ c1 ∨ (x1,g1,c1,b1) ≠ (x2,g2,c2,b2)
    then
225     H[i][p][q].Write(x,g,c,b)
226     c' := ReadCtr_q[i].Read()
227     A[i].Write(x,g,p)
228     H'[i][p][q].Write(x,g,c',b)
229 else
230     reserve(g1)
231     updateQ(i,g1)
232     A[i].Write(x,g,p)
233 toggle_p[i][q] := 1 − b
234 (hCtr_p[i]++) mod n

Operation TRead_p(i)
235 c := (ReadCtr_p[i].Read() + 1) mod 2n
236 H[i][c mod n][p].Write(⊥,⊥,⊥,⊥)
237 H'[i][c mod n][p].Write(⊥,⊥,⊥,⊥)
238 ReadCtr_p[i].Write(c)
239 (x,g,q) := A[i].Read()
240 retv := (x,g)
241 announce(g)
242 (x1,g1,c1,b1) := H[i][q][p].Read()
243 (x2,g2,c2,b2) := H'[i][q][p].Read()
244 if c = c1 ∧ (x1,g1,c1,b1) = (x2,g2,c2,b2)
    then
245     retv := (x1,g1)
246     unannounce(g)
247     announce(g1)
248 return retv
```

Figure 7.11: Revised Implementation of an (m,τ)-TRA object

`reserve(g)` and `unreserve(g)`, instead of `Protect(g)` respectively `Unprotect(g)`.

The second change is that we introduce a new variable $\sigma(p,x)$, for each process p and each block b_x that p owns. When process p resets the value of $Act_q[x]$, for some q, in line 194 of a `GetFree()` operation, we add the current value of $Act_q[x]$ to the value stored in $\sigma(p,x)$. The only purpose of this variable is to provide an invariant in our proofs for the sum $\sigma(p,x) + \sum_{q=0}^{n-1} Act_q[x]$.

Obviously, none of these changes affects the behaviour of the algorithm. From now on, we only use the modified pseudocode presented in Figures 7.9–7.11

7.5.2 Transcript and Linearization Points

Consider any transcript Λ generated by the implementation of type ⟦TLSA⟧⟨TRA⟩ presented in ⟦Figure 7.9⟧⟨Figure 7.11⟩ and Figure 7.10, and let Λ start at time 0. We assume without loss of generality that Λ is complete. If not, we can let all incomplete operations run to completion (note that these operations are wait-free), and a linearization of the interpretation of the resulting complete transcript is also a linearization of $\Gamma(\Lambda)$.

Let $H = \Gamma(\Lambda)$. In the following, we assign a point $lin(M)$ to each operation M in H on the ⟦TLSA⟧⟨TRA⟩ object. We let $\mathcal{L}(H)$ be the sequential history obtained from H, by ordering all operations M in H according to their $lin(M)$ values. We associate each operation M in H with the same time t at which $lin(M)$ occurs in Λ. By construction, the time associated with an operation increases strictly with the position of the operation in $\mathcal{L}(H)$. Later in this section, we prove that $lin(M)$ is in fact the linearization point of M, and $\mathcal{L}(H)$ is a linearization of H.

⟦If M is an unsuccessful $\texttt{TSC}(i,\cdot)$ operation (i.e. M returns `false` in any of lines 155 and 160), we define $lin(M) = t_{M@rsp}$. If M is a successful $\texttt{TSC}(i,(x,g))$ operation (i.e. M returns `true` in line 168), $lin(M)$ is the point at which the calling process successfully executes $A[i].\texttt{SC}(x,g)$ at $t_{M@158}$.⟧⟨If M is a $\texttt{TWrite}(i,(x,g))$ operation, we define $lin(M)$ to be the point at which the calling process writes (x,g) into $A[i]$. I.e. $lin(M)$ is either $t_{M@227}$ or $t_{M@232}$ depending on whether the if-condition in line 224 evaluates to true or false.⟩

Now suppose M is a ⟦$\texttt{TLL}(i)$⟧⟨$\texttt{TRead}(i)$⟩ operation by some process p. Operation M is *direct* if p's if-condition in ⟦line 176⟧⟨line 244⟩ of M evaluates to false, and is *indirect* otherwise. For a direct ⟦$\texttt{TLL}(i)$⟧⟨$\texttt{TRead}(i)$⟩ operation M, we define $lin(M) =$⟦$t_{L@173}$⟧⟨$t_{L@239}$⟩, i.e., the point when p ⟦loads⟧⟨reads⟩ the value of $A[i]$. Now suppose M is an indirect ⟦$\texttt{TLL}(i)$⟧⟨$\texttt{TRead}(i)$⟩. Let (x,g) be the return value of M, and let S be a ⟦$\texttt{TSC}(i,(x,g))$⟧⟨$\texttt{TWrite}(i,(x,g))$⟩ operation by some process q during which q writes ⟦(x,g)⟧⟨(x,g,q)⟩ into $A[i]$ during M — we prove later (see Claim 7.14 on p. 203) that such a ⟦$\texttt{TSC}(i,(x,g))$⟧⟨$\texttt{TWrite}(i,(x,g))$⟩ exists. In this case ⟦and if p sets $flag_p[i]$ in line 181 of M⟧, we let $lin(M) = lin(S)$ ⟦and otherwise, we let

$lin(M) = t_{M@179}$].

If M is a `GetFree()` operation, then we let $lin(M) = t_{M@rsp} = t_{M@199}$. If M is a `Release()` operation, then $lin(M) = t_{M@rsp} = t_{M@202}$. Finally, we define $lin(M) = t_{M@inv} = t_{M@rsp}$, for $M \in \{$`Protect()`,`Unprotect()`,`CancelProtect()`$\}$.

Definition of Free and Occupied in a Transcript. We define the terms "active", "protected", "occupied", and "free" in a canonical way for transcript Λ, by inheriting the properties from $\mathcal{L}(H) = \mathcal{L}(\Gamma(\Lambda))$. For example, tag g becomes *active* at point $lin(G)$, where G is a `GetFree()` operation that returns g (and is executed by the owner of this tag), and it becomes *inactive* at $lin(Re)$, where Re is the next `Release(`g`)` by any process.

Consider a `Protect(`g`)` call P by some process q. Let t_ℓ be the linearization point of q's next operation call on the [[TLSA]]⟨TRA⟩ object or $t_\ell = \infty$ if such a point does not exist. Suppose there is a point in $[lin(P), t_\ell)$, such that g is occupied at that point, and let t^* be the first such point. Then P *succeeds* at t^*. A `Protect(`g`)` call P is *effective at* t if P succeeds before t, and if an operation gets invoked in $\Lambda|q$ during $[lin(P), t]$, then the first such operation is not a `CancelProtect()` call. For any point t during Λ, any process q, and any tag g,

- let $\alpha(q,g,t)$ denote the number of [[`TLL()`]]⟨`TRead()`⟩ operations by q with return value $(\cdot, g)$ that are linearized during $[0,t]$,
- let $\theta(q,g,t)$ represent the number of `Protect(`g`)` calls by q that are effective at t, and
- let $\gamma(q,g,t)$ represent the number of `Unprotect(`g`)` calls by q that are linearized during $[0,t]$.

Then $Protected(g,p)$ at t is $\alpha(q,g,t) + \theta(q,g,t) - \gamma(q,g,t)$. Process q protects tag g if q protects g at least once.

A tag is occupied at point t if it is protected by some process, active, or stored in an element of A, and is free otherwise.

A Good Transcript. A transcript Λ is *good* if it satisfies all of the following for any tag g and any process p.

(G1) For any $\texttt{Unprotect}(g)$ operation U by p in Λ, process p protects g throughout $[t_{U@inv}, lin(U))$,

(G2) for any $\texttt{Release}(g)$ operation Re in Λ, tag g is active throughout $[t_{Re@inv}, lin(Re))$,

(G3) for any $[\![\texttt{TSC}(\cdot,(\cdot,g))]\!]\langle\texttt{TWrite}(\cdot,(\cdot,g))\rangle$ operation S in Λ, tag g is occupied throughout $[t_{S@inv}, lin(S))$,

(G4) suppose p executes an operation call M on the $[\![\text{TLSA}]\!]\langle\text{TRA}\rangle$ object right after it executes a $\texttt{Protect}(g)$ call P, then either tag g is occupied at some point during $[lin(P), t_{M@inv})$, or M is a $\texttt{CancelProtect}(g)$,

(G5) in $\Lambda|p$, any $\texttt{CancelProtect}(g)$ appears only immediately after a $\texttt{Protect}(g)$, and

(G6) suppose p executes a $\texttt{Protect}(g)$ call P, then g must have been occupied at some point during $[t, t_{P@inv})$, where t is $lin(P')$ and P' is p's last $\texttt{Protect}()$ call if such a call exists, or otherwise $t = 0$.

Claim 7.2. *Let Λ be a good transcript, and $H = \Gamma(\Lambda)$. Then for any point t, any tag g, and any process p,*

(a) tag g is active at t in $\mathcal{L}(H)$ if and only if g is active at t in Λ,

(b) $Protected(g,p) = k$ at t in $\mathcal{L}(H)$ if and only if $Protected(g,p) = k$ at t in Λ, and

(c) tag g is occupied at t in $\mathcal{L}(H)$ if and only if g is occupied at t in Λ.

Proof. Suppose tag g becomes active at some point t in Λ. Then there exists a $\texttt{GetFree}()$ operation M in Λ that returns g, and $t = lin(M)$. Operation M appears at point t in $\mathcal{L}(H)$. Thus, g becomes active at point t in $\mathcal{L}(H)$. Next, suppose tag g becomes inactive at some point t in Λ. Then there exists a $\texttt{Release}(g)$ operation M in Λ, such that $t = lin(M)$. Operation

M appears at point t in $\mathcal{L}(H)$. Therefore, g becomes inactive at point t in $\mathcal{L}(H)$. This proves part (a). With essentially the similar arguments, it is easy to see that (b) and (c) are also true. □

Claim 7.3. *If Λ is not a good transcript, then $H = \Gamma(\Lambda)$ has a linearization.*

Proof. Since Λ is not a good transcript, there is an operations call in Λ that does not satisfy one of properties (G1)-(G6) of the good transcript. Let M, executed by some process p, be the first such operation call, and let t be the first point during M such that the subtranscript Λ_t of Λ before this point is a good transcript. In the following, we show how to obtain a sequential history S_H from $\mathcal{L}(H)$ such that S_H is a linearization of H. The main idea is to shift the linearization point of the offending operation M, in such a way that in the resulting sequential history S_H, one of the constraints **(C1)**-**(C7)** is not satisfied. As a result, S_H is trivially valid.

Suppose at point t in Λ, one of properties (G1), (G2), and (G3) is not satisfied. Then for some tag g, at point $t \in [t_{M@inv}, lin(M))$, p does not protect g and M is an $\texttt{Unprotect}(g)$ call, or g is not active and M is a $\texttt{Release}(g)$ call, or g is not occupied and M is a $[\![\texttt{TSC}(\cdot,(\cdot,g))]\!]\langle\texttt{TWrite}(\cdot,(\cdot,g))\rangle$ call. By Claim 7.2 for Λ_t, at point t in $\mathcal{L}(H)$, p does not protect g and M is an $\texttt{Unprotect}(g)$ call, or tag g is not active and M is a $\texttt{Release}(g)$ call, or g is not occupied and M is a $[\![\texttt{TSC}(\cdot,(\cdot,g))]\!]\langle\texttt{TWrite}(\cdot,(\cdot,g))\rangle$ call. We let S_H be the same sequential history as $\mathcal{L}(H)$, except that in S_H, we let M linearize at point t, which is a point during the execution of M. Therefore at point t in S_H, or tag g is not protected when p executes its $\texttt{Unprotect}(g)$ call M, tag g is not active when p executes its $\texttt{Release}(g)$ call M, or g is not occupied when p executes its $[\![\texttt{TSC}(\cdot,(\cdot,g))]\!]\langle\texttt{TWrite}(\cdot,(\cdot,g))\rangle$ call M. Hence, **(C1)**, **(C2)**, or **(C3)** is violated for operation M in S_H, and so S_H is valid.

Next suppose at point t, (G4) is violated. Then for some tag g, M is a $\texttt{Protect}(g)$ call, and throughout $[lin(M), t_{M'@inv})$, g is not occupied, where $M' \neq \texttt{CancelProtect}(g)$ is the next operation invoked in $\Lambda|p$ after M. Thus $t = t_{M'@inv}$. By Claim 7.2, g is not occupied throughout $[lin(M), t)$ in $\mathcal{L}(H)$. We construct S_H exactly in the same way as $\mathcal{L}(H)$, except that we let

M' linearize at $t = t_{M'@inv}$. Thus in S_H, p executes its `Protect(g)` call M at $lin(M)$, and its next operation call is M' at t, and g is not occupied throughout $[lin(M), t)$. Thus, M does not succeed, and it is also not followed by a `CancelProtect(g)` call by p. This violates **(C5)**, and so S_H is valid.

Suppose at point t, property (G5) is not satisfied. Then for some tag g, M is a `CancelProtect(g)` call, and the operation call M' that precedes M in $\Lambda|p$ is not a `Protect(g)` call. Since both M and M' are operation calls by p, they do not overlap, and so M appears before M' in $\mathcal{L}(H)$. We let $S_H = \mathcal{L}(H)$, and then in this history process p executes its `CancelProtect(g)` call M after its operation call M', which is not a `Protect(g)` call. Hence, constraint **(C6)** is not satisfied and so S_H is valid.

Finally, suppose at point t, property (G6) is violated. Then for some tag g, M is a `Protect(g)` call, $t = t_{M@inv}$, and g is not occupied throughout $[t^*, t)$, where t^* is $lin(M')$ and M' is p's last `Protect(g)` call before M if such a call exists, or otherwise $t^* = 0$. By Claim 7.2, tag g is not occupied throughout $[t^*, t)$ in $\mathcal{L}(H)$. We construct S_H in the same way as $\mathcal{L}(H)$, except that we linearize M at $t = t_{M@inv}$. If Λ contains a `Protect(g)` call M' by p before M, then in S_H, tag g is not occupied at any point between M' and M. If M' does not exists in Λ, then in S_H, g is not occupied at any point before the execution of M. Hence, **(C7)** is not satisfied, and so S_H is valid. □

Linearizability Proof Sketch. In order to prove linearizability of the implementation of ⟦TLSA⟧⟨TRA⟩ presented in ⟦Figure 7.9⟧⟨Figure 7.11⟩ and Figure 7.10, we consider a transcript Λ that is obtained from an execution of this implementation. If Λ is not good, then by Claim 7.3, the interpretation of Λ has a linearization. If Λ is a good transcript, we show that ordering operations M by $lin(M)$ yields a valid sequential history (see Lemma 7.33 for more details).

To that end, it is required to show that $lin(M)$ is a point during $[t_{M@inv}, t_{M@rsp}]$, for any operation M. This immediately follows from the definition of $lin(M)$ if M is not an indirect ⟦`TLL()`⟧⟨`TRead()`⟩ operation. In Claim 7.14, we show that if M is an indirect ⟦`TLL()`⟧⟨`TRead()`⟩

operation, then $lin(M)$ is during $[t_{M@inv}, t_{M@rsp}]$.

Lemma 7.4 below implies that for any good transcript Λ, $\mathcal{L}(\Gamma(\Lambda))$ is valid, as we explain after the lemma statement.

Lemma 7.4. *If Λ is a good transcript, then*

(a) the value of $A[i]$ changes to (x,g) at some point t if and only if $t = lin(S)$, where S is a ⟦successful $\texttt{TSC}(i,(x,g))$⟧⟨$\texttt{TWrite}(i,(x,g))$⟩ in Λ,

(b) if a ⟦$\texttt{TLL}(i)$⟧⟨$\texttt{TRead}(i)$⟩ operation L in Λ returns (x,g), then $A[i] = (x,g)$ at $lin(L)$,

(c) if a $\texttt{GetFree()}$ operation G in Λ returns g, then g is not occupied immediately before $lin(G)$ in Λ, and

(d) ⟦a $\texttt{TSC}(i,\cdot)$ operation S succeeds in Λ if and only if there is no successful $\texttt{TSC}(i,\cdot)$ operation S', such that $lin(S') \in (lin(L), lin(S))$, where L is the last $\texttt{TLL}(i)$ by the same process before S in Λ.⟧

As a result of this theorem, we can show the following.

Corollary 7.5. *For every good transcript Λ, $\mathcal{L}(\Gamma(\Lambda))$ is a valid sequential history.*

Proof. We show that each of properties **(S1)**-**(S8)** is satisfied for $\mathcal{L}(\Gamma(\Lambda))$. Property **(S2)** immediately follows Claim 7.2 and the definition of active in a transcript. Property **(S1)** is ensured by Claim 7.2, the same definition for active, and Lemma 7.4(c). Property **(S3a)** follows from Part (a), and **(S3b)** is ensured by Parts (a) and (d) of the same lemma. Properties **(S4a)** and **(S4b)** follow from Part (b). The definition of when a tag is protected in a transcript and Claim 7.2 immediately guarantees **(S5)** and **(S6)**. That definition also implies that a $\texttt{Protect}(g)$ call by some process p only increases the number of times tag g is protected by p, if that call succeeds before p's next operation call on the object linearizes, and only if that operation is not a $\texttt{CancelProtect}(g)$ call. Thus, **(S7)** follows. Now consider a $\texttt{Protect}(g)$ call P by p, and assume p's next operation is a $\texttt{CancelProtect}(g)$ call C. The definition of when a tag is

protected in a transcript implies that the number of times tag g is protected by p before $lin(P)$ is the same as after C is invoked, and C also does not change the number of times tag g is protected by p. Thus, by Claim 7.2, **(S8)** follows. □

To prove Lemma 7.4, we need to establish some guarantees of the algorithm for any good transcript Λ, as it comes in the following. For ease of explanation, from now on we call $lin(M)$ the linearization point of operation M.

More Definitions. As explained earlier, in order to distinguish between $Emp_p[x]$ being modified in a $[\![$`TSC()`$]\!]\langle$`TWrite()`$\rangle$ operation and when it being modified in a $[\![$`TLL()`$]\!]\langle$`TRead()`$\rangle$ or a `Protect()`/`Unprotect()` operation, we introduced operations `reserve()`/`unreserve()` and `announce()`/`unannounce()`.

Consider a point t, a process p, and a tag g. At t, tag g is *reserved* k times by p for the i-th element of the taggable array, if there is an integer ℓ such that in $[0,t)$, process p executed `reserve(`g`)` (from a $[\![$`TSC(`$i,\cdot$`)`$]\!]\langle$`TWrite(`$i,\cdot$`)`$\rangle$) $k+\ell$ times, and `unreserve(`g`)` (from a $[\![$`TSC(`$i,\cdot$`)`$]\!]\langle$`TWrite(`$i,\cdot$`)`$\rangle$) ℓ times. The number of times that tag g is reserved by p for the i-th element of the taggable array at each point in time is denoted by $Rsrv(g,p,i)$. Tag g is *announced* k times by process p at point t if there is an integer ℓ such that in $[0,t)$, process p executed `announce(`g`)` $k+\ell$ times and `unannounce(`g`)` ℓ times. This number is denoted by $Annc(g,p)$.

Tag g is reserved by p for the i-th element of the taggable array, if $Rsrv(g,p,i) > 0$, and g is reserved by p, if there is some $i \in \{0,\ldots,m-1\}$, such that $Rsrv(g,p,i) > 0$. Similarly, tag g is announced by p, if $Annc(g,p) > 0$. A tag g is reserved, if there is a process p, and an index i, such that $Rsrv(g,p,i) > 0$, and g is announced if there is a process p, such that $Annc(g,p) > 0$. Finally, for a block b_x, we let $NumOfActive(x)$ denote the number of tags in block b_x that are active.

Consider a block b_x that is owned by p. We say block b_x is **free**, whenever

$$\boxed{\begin{array}{l}\sigma(p,x) + \sum_{q=0}^{n-1} Act_q[x] = 0, \text{ and} \\ \forall q \in \{0,\ldots,n-1\} : Emp_q[x] = 0\end{array}} \tag{7.7}$$

We later prove (in Claim 7.23) that whenever a block is free, then all tags in that block are free.

Consider an interval I that starts when p executes line 185 for $\rho_p = 0$ and ends at the first point when it increments ρ_p to $2n$ in line 189. It follows immediately from lines 190–197 that variable j_p does not change throughout I. Hence, during I, process p reads (in lines lines 185 and 186) all array entries of $Emp_z[p\beta + j_p]$ and $Act_z[p\beta + j_p]$ for $z \in \{0,\ldots,n-1\}$ twice. Therefore, we call I a **search interval** of block b_x by p, where $x = p\beta + j_p$.

Now consider an interval I' that starts when p executes line 194 for $\rho_p = 2n$, and ends at the first point when it increments ρ_p to 0 in line 189. Again from lines 190–197, it follows that variable j_p does not change throughout I'. During this interval p writes 0 into all array entries $Act_z[p\beta + j_p]$ for $z \in \{0,\ldots,n-1\}$ (in line 194). Therefore, we call I' a **cleaning interval** of block b_x by p, where $x = p\beta + j_p$. Note that in line 196 of a `GetFree()` operation by p during which a cleaning interval of b_x by p ends, p changes the value of the tag that it is about to return to the first tag of block b_x.

7.5.3 Reservation Mechanism

Let p be a process and g^* a tag. In Claim 7.6 below, we show that if p's queue $rsrvQ_p[i]$ contains k copies of g^*, for some $i \in \{0,\ldots,m-1\}$, then $Rsrv(g^*,p,i) \geq k$. Moreover, in Claim 7.7, we show that if at some point $Rsrv(g^*,p,i) = k$ and p is not poised to execute ⟦lines 156–160 of a `TSC(i,·)`⟧⟨lines 219–221 of a `TWrite(i,·)`⟩, then $rsrvQ_p[i]$ contains k copies of g^* at this point. Invariant 7.8 states that $rsrvQ(g^*,p,i)$ is always at least 0, for any process p, any tag g^*, and any $i \in \{0,\ldots,m-1\}$.

Claim 7.6. *If p's local queue $rsrvQ_p[i]$ contains k copies of g^*, for some $i \in \{0,\ldots,m-1\}$, then $Rsrv(g^*,p,i) \geq k$.*

Proof. Let Λ be a transcript on an object T, where T is either of type TLSA or of type TRA.

Case I: T is of type TLSA. Process p calls `reserve(`g^*`)` and `unreserve(`g^*`)` only in a `TSC()` operation. Moreover, $Rsrv(g^*,p,i)$ denotes the number of times tag g^* is reserved by p for the i-th element of the taggable array. Therefore, we consider only `TSC(`$i,\cdot$`)` operations by p. Let $\mathcal{S}$ and $\mathcal{S}'$ be the set of all successful respectively unsuccessful `TSC(`$i,\cdot$`)` operations that p invokes before t.

During an unsuccessful `TSC(`$i,\cdot$`)`, process p can execute `reserve(`g^*`)` only in line 156, however this is followed by an `unreserve(`g^*`)` operation in line 159. Moreover, process p does not modify its local queue $rsrvQ_p[i]$ during an unsuccessful `TSC(`$i,\cdot$`)`. Hence,

$$\text{if } p \text{ calls } \texttt{reserve(}g^*\texttt{)}\ \ell_1 \text{ times in operations in } \mathcal{S}' \text{ during } [0,t], \text{ then } p \text{ calls } \texttt{unreserve(}g^*\texttt{)} \text{ at most } \ell_1 \text{ times in operations in } \mathcal{S}' \text{ during this interval.} \tag{7.8}$$

During each successful `TSC(`$i,\cdot$`)`, process p can execute `reserve(`g^*`)` in lines 156 and 165. If p calls `reserve(`g^*`)` in line 156 (similarly line 165), then it calls an `updateQ(`i,g^*`)` in line 161 (respectively line 166). During each `updateQ(`i,g^*`)`, p enqueues a copy of g^* into $rsrvQ_p[i]$ (in line 203). Assume p enqueues $k+\ell_2$ copies of g^* into $rsrvQ_p[i]$ during $[0,t]$. Then

$$p \text{ calls } \texttt{reserve(}g^*\texttt{)} \text{ at least } k+\ell_2 \text{ times in operations in } \mathcal{S} \text{ during } [0,t]. \tag{7.9}$$

Since p's local queue $rsrvQ_p[i]$ contains k copies of g^* at t, process p dequeued ℓ_2 copies of g^* from $rsrvQ_p[i]$ during $[0,t]$. Process p only dequeues elements from $rsrvQ_p[i]$ in line 204 of an `updateQ(`$i,\cdot$`)`. Right after p dequeues g^* in line 204, it calls `unreserve(`g^*`)` in line 205. Therefore,

$$p \text{ calls } \texttt{unreserve(}g^*\texttt{)} \text{ at most } \ell_2 \text{ times in operations in } \mathcal{S} \text{ during } [0,t]. \tag{7.10}$$

By (7.8), (7.9) and (7.10), we conclude that during interval $[0,t]$, process p calls `reserve`(g^*) at least $k+\ell_1+\ell_2$ times and `unreserve`(g^*) at most $\ell_1+\ell_2$ times in operations in $\mathcal{S}'$ and $\mathcal{S}$, and so $Rsrv(g^*,p,i) \geq k+\ell_1+\ell_2-(\ell_1+\ell_2) = k$ at t.

Case II: T is of type TRA. Process p only calls `reserve`(g^*) and `unreserve`(g^*) in a `TWrite()` operation. Moreover, $Rsrv(g^*,p,i)$ denotes the number of times tag g^* is reserved by p for the i-th element of the taggable array. Therefore, we consider only `TWrite`$(i,\cdot)$ operations by p. Let $\mathcal{S}$ be the set of all `TWrite`$(i,\cdot)$ operations that p invokes before t.

During each `TWrite`$(i,\cdot)$, process p can execute `reserve`(g^*) in line 219 and line 230. If p calls `reserve`(g^*) in line 219 (similarly line 230), then it immediately calls an `updateQ`(i,g^*) in line 222 (respectively line 231). During each `updateQ`(i,g^*), p enqueues a copy of g^* into $rsrvQ_p[i]$ (in line 203). Assume p enqueues $k+\ell$ copies of g^* into $rsrvQ_p[i]$ during $[0,t]$, for some $\ell \geq 0$. Then

$$p \text{ calls } \texttt{reserve}(g^*) \text{ at least } k+\ell \text{ times in operations in } \mathcal{S} \text{ during } [0,t]. \tag{7.11}$$

As p's local queue $rsrvQ_p[i]$ contains k copies of g^* at t, p dequeued ℓ copies of g^* from $rsrvQ_p[i]$ during $[0,t]$. Process p only dequeues elements from $rsrvQ_p[i]$ in line 204 of an `updateQ`$(i,\cdot)$. Right after p dequeues g^* in line 204, it calls `unreserve`(g^*) in line 205. Therefore,

$$p \text{ calls } \texttt{unreserve}(g^*) \text{ at most } \ell \text{ times in operations in } \mathcal{S} \text{ during } [0,t]. \tag{7.12}$$

By (7.11) and (7.12), we conclude that $Rsrv(g^*,p,i) \geq k+\ell-\ell = k$ at t. □

Claim 7.7. *Suppose $Rsrv(g^*,p,i) = k$ at some point t, for some $i \in \{0,\ldots,m-1\}$, and p is not poised to execute ⟦lines 156–160 of a `TSC`$(i,\cdot)$⟧⟨lines 219–221 of a `TWrite`$(i,\cdot)$⟩ operation when p's local variable g has value g^*. Then, p's local queue $rsrvQ_p[i]$ contains at least k copies of g^* at this point.*

Proof. Let Λ be a transcript on an object T, where T is either of type TLSA or of type TRA.

Case I: T is of type TLSA. Since $Rsrv(g^*, p, i) = k$ at point t, there exists an integer ℓ, such that during $[0, t]$, process p calls `reserve(`g^*`)` in `TSC(`$i, \cdot$`)` operations in total $k + \ell$ times, and `unreserve(`g^*`)` in total ℓ times.

At point t, process p is not poised to execute lines 156–160 of a `TSC(`$i, \cdot$`)` operation when p's local variable g has value g^*. Therefore each execution of `reserve(`g^*`)` by p called during $[0, t]$ in an unsuccessful `TSC(`$i, \cdot$`)` is followed by an execution of `unreserve(`g^*`)` prior to t.

Operations `reserve(`g^*`)` and `unreserve(`g^*`)` can only be called in a `TSC()` operation call. Therefore, there exist integers ℓ_1 and ℓ_2, where $\ell_1 + \ell_2 = \ell$, and during $[0, t]$, process p calls

(a) `reserve(`g^*`)` ℓ_1 times in line 156 of an unsuccessful `TSC(`$i, \cdot$`)`,

(b) `unreserve(`g^*`)` ℓ_1 times in line 159 of an unsuccessful `TSC(`$i, \cdot$`)`,

(c) `reserve(`g^*`)` $k + \ell_2$ times in line 156 or line 165 of a successful `TSC(`$i, \cdot$`)`, and

(d) `unreserve(`g^*`)` ℓ_2 times in line 205 of an `updateQ(`$i, \cdot$`)` called in a successful `TSC(`$i, \cdot$`)`.

This implies as we explain in the following, that during $[0, t]$, process p enqueues exactly $k + \ell_2$ copies of g^* into p's queue $rsrvQ_p[i]$ and dequeues exactly ℓ_2 copies of g^* from this queue.

Queue $rsrvQ_p[i]$ is local to p and is only modified during a successful `TSC(`$i, \cdot$`)` operation by p. Process p is not poised to execute lines 156–160 of a `TSC(`$i, \cdot$`)` at t. Thus during $[0, t]$, each time p calls `reserve(`g^*`)` (in line 156, respectively line 165) in a successful `TSC(`$i, \cdot$`)` operation, it also calls `updateQ(`g^*`)` (in line 161, respectively immediately in line 166). Thus during this interval, with every call of `reserve(`g^*`)` in a successful `TSC(`$i, \cdot$`)` operation, p enqueues g^* into its queue $rsrvQ_p[i]$. Moreover during $[0, t]$, each call of `unreserve(`g^*`)` (in line 205 of an `updateQ(`$i, \cdot$`)` called) in a successful `TSC(`$i, \cdot$`)` is immediately preceded by p dequeuing a copy of g^* from its queue $rsrvQ_p[i]$ (in line 204). Therefore, Parts (c)-(d) imply that p's queue $rsrvQ_p[i]$ contains exactly k copies of g^* at t.

Case II: T is of type TRA. Since $Rsrv(g^*, p, i) = k$ at t, there exists an integer ℓ, such that during $[0, t]$, process p calls `reserve(`g^*`)` in `TWrite(`$i, \cdot$`)` operations in total $k + \ell$ times, and

`unreserve`(g^*) in total ℓ times.

Queue $rsrvQ_p[i]$ is local to p and is only modified during a `TWrite`$(i,\cdot)$ operation by p. At point t, process p is not poised to execute lines 219–221 of a `TWrite`$(i,\cdot)$ operation when p's local variable g has value g^*. Therefore during $[0,t]$, each execution of `reserve`(g^*) by p called (in line 219, respectively line 230) is followed by an execution of `updateQ`(g^*) (in line 222, respectively line 231). Thus with every call of `reserve`(g^*) call during $[0,t]$, p enqueues a copy of g^* into its queue $rsrvQ_p[i]$. Hence, p enqueues $k+\ell$ copies of g^* to $rsrvQ_p[i]$ during $[0,t]$. Moreover during $[0,t]$, each call of `unreserve`(g^*) by p (in line 205) of an `updateQ`$(i,\cdot)$ called) in a `TWrite`$(i,\cdot)$ operation is immediately preceded by p dequeuing a copy of g^* from its queue $rsrvQ_p[i]$ (in line 204). Thus, p dequeues ℓ copies of g^* from its queue $rsrvQ_p[i]$ during $[0,t]$. Therefore, p's queue $rsrvQ_p[i]$ contains exactly k copies of g^* at t. □

Invariant 7.8. *At each point in time* $Rsrv(g^*,p,i) \geq 0$.

Proof. Suppose that the invariant holds throughout $[0,t)$ for some t, and at t process p takes one step. We show that the invariant holds right after t. The invariant can only become false, if p executes the atomic operation `unreserve`(g^*) at t, and as a result $Rsrv(g^*,p,i)$ changes to a negative value. Process p calls `unreserve`(g^*) ⟦either in line 159 of an unsuccessful `TSC`$(i,\cdot)$ operation S, or⟧ in line 205 of an `updateQ`$(i,\cdot)$ operation U called from a ⟦`TSC`$(i,\cdot)$⟧⟨`TWrite`$(i,\cdot)$⟩.

⟦First consider the case that at t, process p executes `unreserve`(g^*) in line 159 of S. By the implementation, p has executed a `reserve`(g^*) in line 156 of S, and since we assumed $Rsrv(g^*,p,i) \geq 0$ before t, and therefore before line 156 of S, we have $Rsrv(g^*,p,i) > 0$ right after p executes line 156 of S. The value of $Rsrv(g^*,p,i)$ does not change after this point until p executes `unreserve`(g^*) in line 159 of S at t. Thus, we have $Rsrv(g^*,p,i) \geq 0$ right after t.

Next consider the case that at t, process p executes `unreserve`(g^*) in line 205 of U called from a `TSC`$(i,\cdot)$⟧. At the same point t, process p dequeus g^* from $rsrvQ_p[i]$ in line 204 of U. Thus, right before t, $rsrvQ_p[i]$ contains a copy of tag g^*. By Claim 7.6, we have $Rsrv(g^*,p,i) > 0$ right before t. Thus after executing `unreserve`(g^*) at point t, we have

$Rsrv(g^*,p,i) \geq 0$. □

Next we prove that when $Rsrv(g^*,p,i) > 0$, then the tag remains reserved until p has executed $n+1$ additional successful `TSC`$(i,\cdot)$ operations.

Claim 7.9. *Let* `OP` *represent the set of all* ⟦*successful* `TSC`$(i,\cdot)$⟧⟨`TWrite`$(i,\cdot)$⟩ *operations, for some* $i \in \{0,\ldots,m-1\}$*, by some process* p *in* Λ*. Also let* S_0 *be an operation in* `OP`*, during which* p *executes* `reserve`(g^*) *at some point* t*. If* `OP` *contains* $n+1$ *operations* $S_1,\ldots,S_{n+1}$ *that* p *executes after* S_0*, then* $Rsrv(g^*,p,i) > 0$ *throughout* $[t,t_{S_{n+1}@rsp}]$*, and otherwise throughout* $[t,\infty)$*.*

Proof. At point t, process p executes `reserve`(g^*) in either ⟦line 156 or line 165⟧⟨line 219 or line 230⟩ of S_0. Then it executes `updateQ`(i,g^*) in ⟦line 161 respectively line 166⟧⟨line 222 respectively line 231⟩ of S_0, at some later point $t' > t$. (Operation `updateQ`(i,g^*) contains only one shared memory step.) ⟦Note that since S_0 is a successful `TSC`$(i,\cdot)$ operation, p does not execute `unreserve`(g^*) during $[t,t')$.⟧ Invariant 7.8 implies that $Rsrv(g^*,p,i) \geq 0$ right before t. Therefore throughout $[t,t')$, $Rsrv(g^*,p,i) > 0$.

Now we prove that $Rsrv(g^*,p,i) > 0$ throughout $[t',t_{S_{n+1}@rsp}]$ if `OP` contains $n+1$ operations $S_1,\ldots,S_{n+1}$ that p executes after S_0, and otherwise, throughout $[t',\infty]$. Initially $rsrvQ_p[i]$ contains $2n+4$ elements. During each `updateQ`$(i,\cdot)$ with one atomic operation, one element is enqueued and one element is dequeued from $rsrvQ_p[i]$ (lines 203 and 204). This queue is not modified in any other operation. Hence, the size of $rsrvQ_p[i]$ is always $2n+4$.

During each operation in `OP`, process p calls `updateQ`$(i,\cdot)$ once in ⟦line 161⟧⟨line 222⟩, and it might call it for the second time in ⟦line 166⟧⟨line 231⟩. ⟦During an unsuccessful `TSC`$(i,\cdot)$ it does not call `updateQ()` at all.⟧ Thus, at least one and at most two elements get dequeued from $rsrvQ_p[i]$ during every operations in `OP`⟦, and the queue is not modified during an unsuccessful `TSC`$(i,\cdot)$⟧. Hence, once p enqueues g^* into $rsrvQ_p[i]$ at t', g^* remains in the queue until p completes S_{n+1}, if `OP` contains $n+1$ operations $S_1,\ldots,S_{n+1}$ that p executes after S_0, and

otherwise, throughout $[t',\infty]$. Thus, by Claim 7.6, $Rsrv(g^*,p,i) > 0$ throughout $[t',t_{S_{n+1}@rsp}]$, if such operations $S_1,\ldots,S_{n+1}$ exist, and otherwise throughout $[t',\infty]$. □

Next, we use Claim 7.9 to show that as long as a tag is stored in a register (Corollary 7.10) or a process has a tag provided as a hint (Claim 7.11), that tag is reserved.

Corollary 7.10. *If $A[i] = (x^*,g^*)$ at some point t, then there exists a process p, such that $Rsrv(g^*,p,i) > 0$ at t.*

Proof. Since $A[i] = (x^*,g^*)$ at t, at some point t' for the last time before t, some process p executes ⟦a successful $A[i]$.SC(x^*,g^*) in line 158 of a successful TSC$(i,\cdot)$⟧⟨an $A[i]$.Write(x^*,g^*) either in line 227 or line 232 of a TWrite$(i,\cdot)$⟩ operation call S. Thus, p does not complete another ⟦a successful TSC$(i,\cdot)$⟧⟨TWrite$(i,\cdot)$⟩ call during $[t',t]$, since otherwise, the value of $A[i]$ would change. At ⟦$t_{S@156} < t'$⟧⟨$t_{S@219} < t'$⟩, process p executes reserve(g^*). Thus by Claim 7.9, $Rsrv(g^*,p,i) > 0$ throughout $[t',t]$. □

Claim 7.11. *Consider two processes p and q, and let* OP *represent the set of all ⟦successful* TSC$(i,\cdot)$*⟧⟨*TWrite$(i,\cdot)$*⟩ operations by p. Let S_0 be an operation in* OP*, during which p writes ⟦(x^*,g^*,p) into $H[i][q]$ in line 163⟧⟨(x^*,g^*,c^*,b^*) into $H[i][p][q]$ in line 225⟩ at some point t. ⟨Moreover, assume that p writes (x^*,g^*,c^*,b^*) into $H'[i][p][q]$ in line 228 of S_0⟩. Suppose at point t' ⟦the value stored in $H[i][q]$ changes⟧⟨p writes to either $H[i][p][q]$ or $H'[i][p][q]$⟩ for the first time after t. Then $Rsrv(g^*,p,i) > 0$ throughout $[t,t']$.*

Proof. We have ⟦$t = t_{S_0@163}$⟧⟨$t = t_{S_0@225}$⟩. During S_0, process p must have executed reserve(g^*) in ⟦line 156⟧⟨line 219⟩ at some point before t. If OP contains fewer than $n+1$ operations executed by p after S_0, then by Claim 7.9, we have $Rsrv(g^*,p,i) > 0$ throughout $[t,\infty]$ and so the claim follows. Otherwise, let $S_1,S_2,\ldots$ be the operations in OP that p executes after S_0 in this order. Then by Claim 7.9, we have

$$Rsrv(g^*,p,i) > 0 \text{ throughout } ⟦[t_{S_0@156},t_{S_{n+1}@rsp}]⟧⟨[t_{S_0@219},t_{S_{n+1}@rsp}]⟩. \tag{7.13}$$

Assume for the sake of contradiction that $Rsrv(g^*,p,i) \leq 0$ for the first time during at some point $t^* \in [t,t']$. That is, at t^*, process p executes `unreserve`(g^*) ⟦in line 159 of a `TSC`$(i,(\cdot,g^*))$, or⟧ in line 205 of an `updateQ()` operation called from an operation in `OP`. We have $Rsrv(g^*,p,i) > 0$ throughout $[t,t^*)$, and by (7.13) also throughout $[t_{S_0@156}, t_{S_{n+1}@rsp}] \supseteq [t, t_{S_{n+1}@rsp}]$. Hence, $t^* > t_{S_{n+1}@rsp}$.

⟦First assume that at t^*, process p executes `unreserve`(g^*) in line 159 of a `TSC`$(i,(\cdot,g^*))$ operation S. Then $Rsrv(g^*,p,i) = 1$ right before $t^* = t_{S@159}$. Moreover, p executes `reserve`(g^*) in line 156 of S, thus, $Rsrv(g^*,p,i) = 0$ right before $t_{S@156}$. As $t_{S@156} < t_{S@159} = t^*$ and $Rsrv(g^*,p,i) > 0$ throughout $[t,t^*)$, we have $t_{S@156} \leq t = t_{S_0@163} \leq t_{S@159}$. This is a contradiction as both S and S_0 are `TSC`$(i,\cdot)$ operations by p.⟧

Suppose that at t^*, process p executes `unreserve`(g^*) in line 205 of an `updateQ`(i,g) operation called from an operation S in `OP`. Recall that $t^* > t_{S_{n+1}@rsp}$. Thus, S is invoked after S_{n+1} responds. I.e., $S = S_{j+n+1}$, for some $j > 0$. Consider operations $S_{j+1},\ldots,S_{j+n}$. (By definition, they are all operations in `OP`.) Process p increments its local variable $hCtr_p[i]$ modulo n during each operation in `OP` (in ⟦line 167⟧⟨line 234⟩), so p's local variable $hCtr_p[i] = q$ at the invocation of $S_{j'}$, where $j' \in \{j+1,\ldots,j+n\}$. Note that $1 < j+1 \leq j' \leq j+n$, so $S_{j'}$ gets invoked after $t_{S_1@inv}$ and responds before $t_{S_{j+n+1}@inv}$ and so before $t^* \leq t'$. By the assumption of the claim, ⟦the value of $H[i][q]$ does not change throughout $[t,t') = [t_{S_0@163}, t)$, so p loads $(x^*,g^*,p) \neq (\perp,\perp,\perp)$ from $H[i][q]$ in line 157 of $S_{j'}$⟧⟨p does not write to $H[i][p][q]$ or $H'[i][p][q]$ during (t,t'), so its if-condition in line 224 of $S_{j'}$ evaluates to false and p execute lines 230–232 during $S_{j'}$⟩. Therefore, p executes `reserve`(g^*) in ⟦line 165⟧⟨line 230⟩ of $S_{j'}$. By Claim 7.9 (substituting $S_{j'}$ for S_0 in that claim), we have $Rsrv(g^*,p,i) > 0$ throughout ⟦$[t_{S_{j'}@165}, t_{S_{j'+n+1}@rsp}]$⟧⟨$[t_{S_{j'}@230}, t_{S_{j'+n+1}@rsp}]$⟩. Therefore $Rsrv(g^*,p,i) > 0$ at least throughout ⟦$[t_{S_{j+n}@165}, t_{S_{j+n+2}@rsp}]$⟧⟨$[t_{S_{j+n}@230}, t_{S_{j+n+2}@rsp}]$⟩ and therefore throughout the execution of S_{j+n+1} that comprises t^*, which is a contradiction. □

7.5.4 Properties of the LL() Operation

The following observation shows that although the TLL() operation of the TLSA object has a repeat-until loop, it only requires a constant number of steps.

Observation 7.12. *The repeat-until loop in lines 170–172 of any* TLL(i) *operation terminates after at most two iterations.*

Proof. Suppose for the sake of contradiction that some process q executes line 172 more than twice during a TLL(i) operation L. This implies that q's $H[i][q]$.SC() operation in line 172 fails at least twice during L. Therefore, two other $H[i][q]$.SC() operations sc_1 by p_1 and sc_2 by p_2 succeed during the interval that starts when q executes $H[i][q]$.LL() in line 171 for the first time during L and ends when q executes $H[i][q]$.SC($\perp,\perp,\perp$) in line 172 for the second time during L. Since only process $q \notin \{p_1, p_2\}$ can execute an SC() on $H[i][q]$ during a TLL(i) operation, p_j, $j \in \{1,2\}$ executes sc_j in line 163 of a TSC($i,\cdot$) operation. Assume w.l.o.g. that sc_1 is executed before sc_2, and let S_j be the TSC($i,\cdot$) operation during which p_j executes sc_j in line 163. Both, sc_1 and sc_2, are successful SC() operations on $H[i][q]$, therefore sc_1 cannot be executed between the $H[i][q]$.LL() operation in line 157 of S_2 and sc_2. Thus, p_1 executes sc_1 before the $H[i][q]$.LL() operation in line 157 of S_2. Since only process q can write $(\perp,\perp,\perp)$ to $H[i][q]$, and it does not do so between sc_1 and sc_2, and also the value p_1 writes to $H[i][q]$ in sc_1 is $(\cdot,\cdot,p_1) \neq (\perp,\perp,\perp)$, the $H[i][q]$.LL() operation in line 157 of S_2 reads some non-$\perp$ values. Hence, the if-condition in line 162 of S_2 evaluates to `false` and so line 163 of S_2 does not get executed, which is a contradiction. □

Now we state and prove some properties of indirect $[\![$TLL()$]\!]\langle$TRead()$\rangle$ operations, that is, those that return a value that was provided as a hint. More specifically, we show that if some $[\![$TLL(i)$]\!]\langle$TRead(i)$\rangle$ operation by process p returns a pair (x^*,g^*), which p obtained as a hint, then some process p' wrote $[\![(x^*,g^*)]\!]\langle(x^*,g^*,p')\rangle$ to $A[i]$ at some point during p's

$[\![\texttt{TLL}(i)]\!]\langle\texttt{TRead}(i)\rangle$ operation. Moreover, tag g^* is reserved starting from the point at which p' writes to $A[i]$ until the $[\![\texttt{TLL}(i)]\!]\langle\texttt{TRead}(i)\rangle$ operation responds.

Claim 7.13. *Let q^* be a process and $b^* \in \{0,1\}$, and let S_1 and S_2 be two* $\texttt{TWrite}(i,\cdot)$ *operations by some process p, such that S_1 is invoked before S_2. Suppose at the invocation of both S_1 and S_2, $hCtr_p[i] = q^*$ and $toggle_p[i][q^*] = b^*$. Then during $(t_{S_1@rsp}, t_{S_2@inv})$, process p executes at least one* $\texttt{TWrite}(i,\cdot)$ *operation, at the invocation of which $hCtr_p[i] = q^*$.*

Proof. Process p toggles the value of $toggle_p[i][q^*]$ and increments the value of $hCtr_p[i]$ (modulo n) in lines 233–234 of S_1. These local variables are not modified elsewhere. Thus, at the response of S_1, $hCtr_p[i] = q^* + 1 (\mathrm{mod} n)$ and $toggle[i][q^*] = 1 - b^*$. Moreover, $toggle[i][q^*] = b^*$ at the invocation of S_2. Thus, p toggles the value of this register, after the response of S_1 and before the invocation of S_2.

The value of $toggle[i][q^*]$ is only modified in line 233 of a $\texttt{TWrite}(i,\cdot)$ call by p, and only when $hCtr_p[i] = q^*$. Therefore, p must execute at least one $\texttt{TWrite}(i,\cdot)$ operation call between the response of S_1 and the invocation of S_2 such that $hCtr_p[i] = q^*$ just before p executes line 233 and so at the invocation of this operation call. □

Claim 7.14. *Consider an indirect $[\![\texttt{TLL}(i)]\!]\langle\texttt{TRead}(i)\rangle$ operation L by some process p, which returns some value (x^*, g^*). Then some process p' executes a $[\![\texttt{TSC}(i,(x^*,g^*))]\!]\langle\texttt{TWrite}(i,(x^*,g^*))\rangle$ operation S, such that*

(a) process p' $[\![$successfully executes $A[i].\texttt{SC}(x^*,g^*)$ in line 158$]\!]\langle$writes (x^*,g^*,p) to $A[i]$ in line 227$\rangle$ of S,

(b) $[\![lin(S) = t_{S@158} \in [t_{L@inv}, t_{L@175}]]\!]\langle lin(S) = t_{S@227} \in [t_{L@238}, t_{L@243}]\rangle$,

(c) $Rsrv(g^*,p',i) > 0$ throughout $[\![[t_{S@158}, t_{L@rsp}]]\!]\langle[t_{S@227}, t_{L@rsp}]\rangle$, and

(d) let α be the value of $Annc(g^*,p)$ at point $t_{L@inv}$, then $Annc(g^*,p) = \alpha + 1$, throughout $[\![[t_{L@182}, t_{L@rsp}]]\!]\langle[t_{L@247}, t_{L@rsp}]\rangle$.

Proof. Let Λ be a transcript on an object T, where T is either of type TLSA or of type TRA.

Case I: T is of type TLSA. Since L is an indirect TLL() that returns (x^*, g^*), and so the if-condition in line 176 of L evaluates to true, p must have read $(x^*, g^*, p') \neq (\perp, \perp, \perp)$ from $H[i][p]$ in line 175 of L, for some process p'. That process p' must have executed a successful $H[i][p]$.SC(x^*, g^*, p') in line 163 of a TSC$(i, (x^*, g^*))$ operation S, such that $H[i][p] = (x^*, g^*, p')$ throughout $[t_{S@163}, t_{L@175}]$. This implies

$$t_{S@163} < t_{L@175} \tag{7.14}$$

When $H[i][p]$ stores a non-$\perp$ value, only process p can change the value stored in this variable (in line 172 of a TLL(i)), and it does not do so during $[t_{L@175}, t_{L@rsp}]$. Moreover, due to the LL() operation in line 157 and the if-condition in line 162, no process can execute a successful $H[i][p]$ in line 163, when the value of $H[i][p]$ is not $(\perp, \perp, \perp)$. Thus,

$$H[i][p] = (x^*, g^*, p') \text{ throughout } [t_{S@163}, t_{L@rsp}]. \tag{7.15}$$

Consider again operation S during which p' executes a successful $H[i][p]$.SC(x^*, g^*, p') in line 163. Process p' must have executed a successful $A[i]$.SC(x^*, g^*) in line 158 of S, and this proves Part (a). Operation S is a successful TSC(), so $lin(S) = t_{S@158}$. Moreover by (7.14), $t_{S@158} < t_{S@163} \leq t_{L@175}$. So to complete the proof of Part (b), we need to show that $t_{S@158} \geq t_{L@inv}$.

Process p' executes a successful $H[i][p]$.SC(x^*, g^*, p') in line 163 of S. Hence by line 162, the value stored in $H[i][p]$ is $(\perp, \perp, \perp)$ when p' executes line 157 of S and

$$\text{no process writes to } H[i][p] \text{ during } [t_{S@157}, t_{S@163}). \tag{7.16}$$

Now, assume for the sake of contradiction that $t_{S@158} < t_{L@inv}$. Process p writes $(\perp, \perp, \perp)$ to $H[i][p]$ at some point during $[t_{L@inv}, t_{L@175}] \subseteq [t_{S@158}, t_{L@175}]$. However, this contradict (7.15) and (7.16), and so Part (b) follows.

Process p' executes a successful $A[i]$.SC(x^*, g^*) at $t_{S@158}$. At $t_{S@156} < t_{S@158}$, process p executes reserve(g^*). Thus by Claim 7.9, $Rsrv(g^*, p', i) > 0$ starting at $t_{S@156}$ until it

completes $n+1$ additional successful $\texttt{TSC}(i,\cdot)$ operations, and so throughout $[t_{S@158}, t_{S@rsp}]$. Moreover, by (7.15) and Claim 7.11, $Rsrv(g^*,p,i) > 0$ throughout $[t_{S@163}, t_{L@rsp}]$, and this completes the proof of Part (c).

To prove Part (d), recall that p reads (x^*,g^*,p') from $H[i][p]$ in line 175 of L, so it executes $\texttt{announce}(g^*)$ in line 182 of L. Moreover, if p executes $\texttt{announce}(g)$ in line 174 of L, for some g, then it also executes $\texttt{unannounce}(g)$ in line 178 of L. As p does not call $\texttt{announce}()$ and $\texttt{unannounce}()$ anywhere else during L, $Annc(g^*,p)$ is one larger throughout $[t_{L@182}, t_{L@rsp}]$ than at $t_{L@inv}$.

Case II: T is of type TRA. Let c^* be the value p writes to $ReadCtr_p[i]$ in line 238 of L. Thus,

$$ReadCtr_p[i] = c^* \text{ throughout } [t_{L@238}, t_{L@rsp}]. \tag{7.17}$$

Since the if-condition in line 244 evaluates to $\texttt{true}$ and L returns (x^*,g^*), p must have read $(x^*,g^*,c^*,b^*) \neq (\perp,\perp,\perp,\perp)$, for some value of $b^* \in \{0,1\}$, from both $H[i][p'][p]$ and $H'[i][p'][p]$ in line 242 respectively line 243 of L, for some $p' \in \{0,\ldots,n-1\}$. Recall that only process p' (in line 225 and line 228 of a $\texttt{TWrite}(i,\cdot)$) and process p (in line 236 and line 237 of a $\texttt{TRead}(i)$) can write to $H[i][p'][p]$ or $H[i]'[p'][p]$, and the only value p can write to one of those variables is $(\perp,\perp,\perp,\perp)$. Thus, process p' must have written (x^*,g^*,c^*,b^*) to $H[i][p'][p]$ and $H'[i][p'][p]$ at some time before $t_{L@242}$ respectively $t_{L@243}$.

Let S and S' be $\texttt{TWrite}(i,\cdot)$ operations by process p', such that:

$$\begin{aligned}&\text{at } t_{S@225}\text{, process } p' \text{ writes } (x^*,g^*,c^*,b^*) \text{ to } H[i][p'][p]\\&\text{for the last time before } t_{L@243}\text{, and}\end{aligned} \tag{7.18}$$

$$\begin{aligned}&\text{at } t_{S'@228}\text{, process } p' \text{ writes } (x^*,g^*,c^*,b^*) \text{ to } H'[i][p'][p]\\&\text{for the last time before } t_{L@243}.\end{aligned} \tag{7.19}$$

(We intentionally choose S such that $t_{S@225}$ happens before $t_{L@243}$ but not necessarily before $t_{L@242}$.)

Process p reads (x^*,g^*,c^*,b^*) from $H'[i][p'][p]$ at $t_{L@243}$, thus by (7.19), we have

$$H'[i][p'][p] = (x^*,g^*,c^*,b^*) \text{ throughout } [t_{S'@228}, t_{L@243}]. \tag{7.20}$$

Moreover, p reads (x^*,g^*,c^*,b^*) from $H[i][p'][p]$ at $t_{L@242}$. By (7.18), we have

$$\text{if } t_{S@225} < t_{L@242}, \text{ then } H[i][p'][p] = (x^*,g^*,c^*,b^*) \text{ throughout } [t_{S@225}, t_{L@242}]. \tag{7.21}$$

Now we prove that $S = S'$. Suppose not. By (7.18) and (7.19), p' writes (x^*,g^*,c^*,b^*) into $H[i][p'][p]$ in line 225 of S and the same quadruple into $H'[i][p'][p]$ in line 228 of S'. This implies that $hCtr_{p'}[i] = p$ and $toggle[i][p] = b^*$ at the invocation of both S and S'. Thus by Claim 7.13,

$$\begin{aligned}&\text{during } (\min(t_{S@rsp}, t_{S'@rsp}), \max(t_{S@inv}, t_{S'@inv})),\ p' \text{ executes at least}\\ &\text{one } \texttt{TWrite}(i,\cdot) \text{ operation, at the invocation of which } hCtr_{p'}[i] = p.\end{aligned} \tag{7.22}$$

Let $S_1^*,\ldots,S_k^*$ be the $\texttt{TWrite}(i,\cdot)$ operation calls executed between S and S', such that $hCtr_{p'}[i] = p$ at the invocation of $S_1^*,\ldots,S_k^*$. By (7.22), $k \geq 1$.

Case II.1: Suppose S happens before S'. By (7.19), this implies that $t_{S@225} < t_{S'@228} < t_{L@243}$. First we prove that

$$t_{S@225} < t_{L@242}. \tag{7.23}$$

Suppose for the sake of contradiction that $t_{S@225} \geq t_{L@242}$. Then $t_{S'@218} > t_{L@242}$, because S happens before S'. This in addition to (7.19) implies that $t_{S'@218} \in [t_{L@242}, t_{L@243}]$. So by (7.17), p' reads c^* from $ReadCtr_p[i]$ in line 218 of S'. Thus, p' writes the same quadruple (x^*,g^*,c^*,b^*) to $H[i][p'][p]$ in line 225 of S' as it writes to $H'[i][p'][p]$ in line 228 (by (7.19)). However, this is a contradiction, since by (7.18) process p' writes (x^*,g^*,c^*,b^*) to $H[i][p'][p]$ for the last time before $t_{L@243}$ during S (which happens before S'). Hence, (7.23) is true.

Next, we prove that

$$t_{S'@inv} < t_{L@238} \tag{7.24}$$

If $t_{S'@inv} > t_{L@238}$, then by (7.19), $t_{S'@inv} \in [t_{L@238}, t_{L@rsp}]$. Thus by (7.17), p' reads c from $ReadCtr_p[i]$ in line 218 of S' at $t_{S'@inv} = t_{S'@218}$. Therefore, since by (7.19), p' writes (x^*, g^*, c^*, b^*) to $H'[i][p'][p]$ at $t_{S'@228}$, p' also writes the same quadruple to $H[i][p'][p]$ at $t_{S'@225}$. This contradicts (7.18), because by (7.23) and the assumption that S happens before S', $t_{S'@225} \in [t_{S@225}, t_{L@242}]$. So (7.24) follows.

Now consider again operations $S_1^*, \ldots, S_k^*$ that all complete during $[t_{S@rsp}, t_{S'@inv}]$. By (7.24), we have $t_{S'@inv} < t_{L@238} < t_{L@242}$. Thus,

$$\text{operations } S_1^*, \ldots, S_k^* \text{ all complete during } [t_{S@225}, t_{L@242}]. \tag{7.25}$$

Case II.1-1: First suppose that during at least one operation S_j^*, $j \in \{1, \ldots, k\}$, the if-condition in line 224 evaluates to true. Thus p' writes to $H[i][p'][p]$ and $H'[i][p'][p]$ (in lines 225–228) of S_j^*. By (7.25), S_j^* completes during $[t_{S@225}, t_{L@242}]$. Thus, p' writes to $H[i][p'][p]$ in line 225 of S_j^* during $[t_{S@225}, t_{L@242}]$. Process p' cannot write (x^*, g^*, c^*, b^*) to $H[i][p'][p]$ during this interval, because this would contradict (7.18). Hence, p' writes a different value to $H[i][p'][p]$ in line 225 of S_j^* during $[t_{S@225}, t_{L@242}]$ This is a contradiction to (7.21) and (7.23).

Case II.1-2: Now suppose that in all $S_1^*, \ldots, S_k^*$, the if-condition in line 224 evaluates to false. Fix an arbitrary integer $j \in \{1, \ldots, k\}$. By the choice of $S_1^*, \ldots, S_k^*$, at the invocation of S_j^*, $hCtr_{p'}[i] = p$. Therefore, p' reads $H[i][p'][p]$ and $H'[i][p'][p]$ in lines 220–221 of S_j^*. But since the if-condition in line 224 of S_j^* evaluates to false, p' reads the same quadruple from both $H[i][p'][p]$ and $H'[i][p'][p]$ in lines 220–221 of S_j^*. By (7.21) and (7.23), $H[i][p'][p] = (x^*, g^*, c^*, b^*)$ throughout $[t_{S@225}, t_{L@242}]$, and so by (7.25) throughout the execution of S_j^*. Thus, p' reads (x^*, g^*, c^*, b^*) from both $H[i][p'][p]$ and $H'[i][p'][p]$ in lines 220–221 of S_j^*. Moreover, since the if-condition in line 224 of S_j^* evaluates to `false`, p' reads c^* from $ReadCtr_p[i]$ in line 218 of S_j^*. Hence

$$ReadCtr_p[i] = c^* \text{ at } t_{S_j^*@218}. \tag{7.26}$$

Operation S_j^* completes during $[t_{S@rsp}, t_{S'@inv}]$, and so by (7.24), before point $t_{L@238}$, at which

p increments (modulo $2n$) the value stored in $ReadCtr_p[i]$ to c^*. Therefore by (7.26), the value of $ReadCtr_p[i]$ wraps around during $[t_{S_j^*@218}, t_{L@238}]$. Since process p increments $ReadCtr_p[i]$ modulo $2n$ only in line 238 of each `TRead`(i) operation call, and this variable is not modified elsewhere, p must execute line 238 at least $2n$ times during $[t_{S_j^*@218}, t_{L@238}]$. Thus, p executes lines 236–237 at least $2n-1$ times during the same interval, and in particular at least once when its local variable $c \bmod n$ has value p'. Hence, p writes $(\perp,\perp,\perp,\perp)$ to $H[i][p'][p]$ at some point during $[t_{S_j^*@218}, t_{L@238}] \subseteq [t_{S@rsp}, t_{L@238}]$. This contradicts (7.21) and (7.23).

Case II.2: Now assume S' happens before S. Thus by (7.18), $t_{S'@228} < t_{S@225} < t_{L@243}$. In the following, we show that

$$t_{S@inv} \geq t_{L@238}. \tag{7.27}$$

Assume for the sake of contradiction that $t_{S@inv} < t_{L@238}$. By (7.18), process p writes (x^*, g^*, c^*, b^*) to $H[i][p'][p]$ in line 225 of S. Thus, p' reads c^* from $ReadCtr_p[i]$ at $t_{S@218} = t_{S@inv}$. At $t_{L@238} > t_{S@inv}$, p increments (modulo $2n$) the value stored in $ReadCtr_p[i]$ to c^*. Since process p increments $ReadCtr_p[i]$ modulo $2n$ only in line 238 of each `TRead`(i) operation call, and this variable is not modified elsewhere, p must execute line 238 at least $2n$ times during $[t_{S@inv}, t_{L@238}]$. Thus, p executes lines 236–237 at least $2n-1$ times during the same interval, and in particular at least once when its local variable $c \bmod n$ has value p'. Hence, p writes $(\perp,\perp,\perp,\perp)$ to $H'[i][p'][p]$ at some point during $[t_{S@inv}, t_{L@238}]$, and since S' happens before S, during $[t_{S'@228}, t_{L@238}]$. This contradicts (7.20). Thus, (7.27) follows.

Now consider again operations $S_1^*, \ldots, S_k^*$ that all complete during $[t_{S'@rsp}, t_{S@inv}]$. By (7.18), we have $t_{S'@228} < t_{S'@rsp} < t_{S@inv} < t_{S@225} < t_{L@243}$. Thus,

$$\text{operations } S_1^*, \ldots, S_k^* \text{ all complete during } [t_{S'@228}, t_{L@243}]. \tag{7.28}$$

Case II.2-1: Suppose that during at least one operation S_j^*, $j \in \{1, \ldots, k\}$, the if-condition in line 224 evaluates to true. Thus p' writes to $H[i][p'][p]$ and $H'[i][p'][p]$ (in lines 225–228) of S_j^*. By (7.28), operation S_j^* completes during $[t_{S'@228}, t_{L@243}]$. However, p' does not write

(x^*, g^*, c^*, b^*) to $H'[i][p'][p]$ during S_j^*, as this would contradict (7.19). Then p' writes a different quadruple, which contradicts (7.20).

Case II.2-2: Now suppose that in all $S_1^*, \ldots, S_k^*$, the if-condition in line 224 evaluates to false. By the choice of S_k^*, at the invocation of S_k^*, $hCtr_{p'}[i] = p$. Therefore, process p' reads $H[i][p'][p]$ and $H'[i][p'][p]$ in lines 220–221 of S_k^*. But since the if-condition in line 224 of S_k^* evaluates to false, p' reads the same quadruple from both $H[i][p'][p]$ and $H'[i][p'][p]$ in lines 220–221 of S_k^*. By (7.20), $H'[i][p'][p] = (x^*, g^*, c^*, b^*)$ throughout $[t_{S'@228}, t_{L@243}]$, and so by (7.28) throughout the execution of S_k^*. Thus, p' reads (x^*, g^*, c^*, b^*) from both $H[i][p'][p]$ and $H'[i][p'][p]$ in lines 220–221 of S_k^*.

Now we show that the value of $H[i][p'][p]$ remains equal to (x^*, g^*, c^*, b^*) throughout $[t_{S_k^*@220}, t_{S@220}]$. Only process p and p' can write to $H[i][p'][p]$. Process p' does not write to $H[i][p'][p]$ after S_k^* and before $t_{S@220}$, because S_k^* is the last `TWrite(`i`,·)` call before S, at the invocation of which $hCtr_{p'}[i] = p$. Since p can only reset the value of this register, suppose for the sake of contradiction that p writes $(\perp, \perp, \perp, \perp)$ to $H[i][p'][p]$ in line 236 of a `TRead(`i`)` operation L', such that $t_{L'@236} \in [t_{S_k^*@220}, t_{S@220}]$. By (7.28), $t_{S'@228} < t_{S_k^*@220}$, and by (7.18), $t_{S@225} < t_{L@243}$. Thus, $t_{L'@236} \in [t_{S'@228}, t_{L@243}]$. This contradicts (7.20). Therefore, $H[i][p'][p] = (x^*, g^*, c^*, b^*)$ at $t_{S@220}$.

Moreover, by (7.18), $t_{S@221} < t_{L@243}$, so by (7.20), $H'[i][p'][p] = (x^*, g^*, c^*, b^*)$ at $t_{S@221} \in [t_{S'@228}, t_{L@243}]$. By (7.18), p' writes (x^*, g^*, c^*, b^*) to $H[i][p'][p]$ in line 225 of S, thus, p' reads c^* from $ReadCtr_p[i]$ in line 218 of S. Therefore, the if-condition in line 224 of S evaluates to `false`, and so p' does not write to $H[i][p'][p]$ during S, which contradicts (7.18). This completes the proof for $S = S'$.

Proof of Part (a). As we just proved $S = S'$, i.e. there exist a `TWrite`$(i,\cdot)$ operation S, such that

$$\text{during } S\text{, process } p' \text{ writes } (x^*,g^*,c^*,b^*) \text{ to } H[i][p'][p] \text{ and } H'[i][p'][p] \text{ at } t_{S@225} \text{ resp. } t_{S@228} \text{ for the last time before } t_{L@243}. \tag{7.29}$$

And therefore, p' writes (x^*,g^*,p) to $A[i]$ in line 227 of S.

Proof of Part (b). As p' writes to $A[i]$ in line 227 of S, we have $lin(S) = t_{S@227}$. By (7.29), we know $t_{S@228} < t_{L@243}$, therefore we have:

$$t_{S@227} < t_{L@243} \tag{7.30}$$

Next we prove that $t_{S@226} \geq t_{L@238}$, and therefore $t_{S@227} > t_{L@238}$, which completes the proof of Part (b).

Suppose for the sake of contradiction that $t_{S@226} < t_{L@238}$. Recall that p' writes (x^*,g^*,c^*,b^*) to $H'[i][p'][p]$ in line 228 of S'. Hence, p' must read c^* from $ReadCtr_p[i]$ at $t_{S@226}$. Thus, $ReadCtr_p[i] = c^*$ at both $t_{S@226}$ and $t_{L@238}$, but not just before $t_{L@238}$, and recall that we assumed $t_{S@226} < t_{L@238}$. Process p increments $ReadCtr_p[i]$ modulo $2n$ in line 238 of each `TRead`(i) operation. Thus, p must execute line 238 at least $2n$ times during $[t_{S@226}, t_{L@238}]$. In particular, p must execute line 236 at least n times during that interval, and specifically at least once when its local variable $c \bmod n = p$. Therefore, p writes $(\perp,\perp,\perp,\perp)$ to $H[i][p'][p]$ at least once during $[t_{S@226}, t_{L@238}]$. As p reads (x^*,g^*,c^*,b^*) from $H[i][p'][p]$ at $t_{L@242}$, p' must write this value to $H[i][p'][p]$ at some point during $[t_{S@226}, t_{L@242}]$, which contradicts (7.29).

Proof of Part (c). By (7.29), we know that p' writes (x^*,g^*,c^*,b^*) to both $H[i][p'][p]$ and $H'[i][p'][p]$ during S. Let S^* be the first `TWrite`$(i,\cdot)$ operation after S by p' during which p' writes to $H[i][p'][p]$ and $H'[i][p'][p]$ at $t_{S^*@225}$ respectively $t_{S^*@228}$. Moreover, by Part (b), we know $t_{S@rsp} > t_{S@227} > t_{L@238}$, and therefore $t_{S^*@inv} > t_{L@238}$.

Note that after S, process p' only writes to $H[i][p'][p]$ and $H'[i][p'][p]$, if either it reads $ReadCtr_p[i] \neq c^*$, or p resets $H[i][p'][p]$ or $H'[i][p'][p]$. Neither the value of $ReadCtr_p[i]$

changes nor p resets any of $H[i][p'][p]$ and $H'[i][p'][p]$ during $(t_{L@238}, t_{L@rsp}]$ and so during $[t_{S^*@inv}, t_{L@rsp}]$. Thus, $t_{S^*@225} > t_{L@rsp}$.

By Claim 7.11, we know that $Rsrv(g^*, p', i) > 0$ throughout $[t_{S@225}, t_{S^*@225}]$. Thus, $Rsrv(g^*, p', i) > 0$ throughout $[t_{S@227}, t_{L@rsp}] \subseteq [t_{S@225}, t_{S^*@225}]$.

Proof of Part (d). Recall that p reads (x^*, g^*, c^*, b^*) from $H[i][p'][p]$ in line 242 of L, so it executes `announce`(g^*) in line 247 of L. Moreover, if p executes `announce`(g) in line 241 of L, for some g, then it also executes `unannounce`(g) in line 246 of L. As p does not call `announce()` and `unannounce()` anywhere else during L, $Annc(g^*, p)$ is one larger throughout $[t_{L@247}, t_{L@rsp}]$ than at $t_{L@inv}$. □

In the previous claim, we discussed about the case in which a $[\![$`TLL()`$]\!]\langle$`TRead()`$\rangle$ returns a value that was provided as a hint. Next we consider a $[\![$`TLL(`i`)`$]\!]\langle$`TRead(`i`)`$\rangle$ operation by some process p which returns some value (x^*, g^*) that the process reads from $A[i]$, i.e. when the $[\![$`TLL(`i`)`$]\!]\langle$`TRead(`i`)`$\rangle$ is direct. In the following claim, we show that in this case tag g^* is reserved until $Annc(g^*, p)$ is incremented during p's $[\![$`TLL(`i`)`$]\!]\langle$`TRead(`i`)`$\rangle$ operation, in addition to some other properties of a direct $[\![$`TLL(`i`)`$]\!]\langle$`TRead(`i`)`$\rangle$.

Claim 7.15. *Consider a direct $[\![$`TLL(`i`)`$]\!]\langle$`TRead(`i`)`$\rangle$ operation L by some process p, which returns some pair (x^*, g^*). Then the following is true for some process p'.*

(a) Process p' completes fewer than n $[\![$successful `TSC(`i`,·)`$]\!]\langle$`TWrite(`i`,·)`$\rangle$ operations during $[\![[t_{L@173}, t_{L@175}]]\!]\langle[t_{L@239}, t_{L@242}]\rangle$,

(b) $Rsrv(g^*, p', i) > 0$ throughout $[\![[t_{L@173}, t_{L@175}]]\!]\langle[t_{L@239}, t_{L@242}]\rangle$, and

(c) let α be the value of $Annc(g^*, p)$ at point $t_{L@inv}$, then $Annc(g^*, p) = \alpha + 1$, throughout $[\![[t_{L@174}, t_{L@rsp}]]\!]\langle[t_{L@241}, t_{L@rsp}]\rangle$.

Proof. Let Λ be a transcript on an object T, where T is either of type TLSA or of type TRA. Operation L is a direct $[\![$`TLL()`$]\!]\langle$`TRead()`$\rangle$ with return value (x^*, g^*). Thus, process p must

read $[\![(x^*,g^*)]\!]\langle(x^*,g^*,p'),$ for some $p' \in \{0,\ldots,n-1\}\rangle$ from $A[i]$ in $[\![$line 173$]\!]\langle$line 239$\rangle$ of L. Therefore, some process p' must have previously $[\![$executed a successful $A[i]$.SC(x^*,g^*) in line 158 of some TSC$(i,(x^*,g^*))]\!]\langle$written (x^*,g^*,p') to $A[i]$ in line 227 or line 232) of some TWrite$(i,(x^*,g^*))\rangle$ operation S.

$\langle$Let c^* be the value p writes to $ReadCtr_p[i]$ at $t_{L@238}$. As p does not change this value during the remaining of its TRead(i) operation, we have

$$ReadCtr_p[i] = c^* \text{ throughout } [t_{L@239}, t_{L@243}]. \tag{7.31}$$

Moreover, by the pseudocode,

$$p \text{ does not write to } H[i][p'][p] \text{ and } H'[i][p'][p] \text{ throughout } [t_{L@239}, t_{L@243}]. \tag{7.32}$$

$\rangle$

Proof of Part (a) if T is of type TLSA. Consider process p', and suppose for the sake of contradiction that p' completes at least n successful TSC$(i,\cdot)$ operations during $[t_{L@173}, t_{L@175}]$. Since p' increments its local variable $hCtr_{p'}[i]$ modulo n during each successful TSC$(i,\cdot)$ operation, $hCtr_{p'}[i] = p$ at the invocation of at least one of these successful TSC$(i,\cdot)$ operations. Let S' be the first such TSC$(i,\cdot)$ operation by p'. Thus,

$$[t_{S'@inv}, t_{S'@rsp}] \subseteq [t_{L@173}, t_{L@175}]. \tag{7.33}$$

Next, we show that there exist x, g and q, such that

$$H[i][p] = (x,g,q) \neq (\perp,\perp,\perp) \text{ throughout } [t_{S'@rsp}, t_{L@rsp}]. \tag{7.34}$$

Then since by (7.33), $t_{S'@rsp} \leq t_{L@175}$, and so $[t_{L@175}, t_{L@rsp}] \subseteq [t_{S'@rsp}, t_{L@rsp}]$, (7.33) implies that $H[i][p] = (x,g,q) \neq (\perp,\perp,\perp)$ at point $t_{L@175}$. However, since L is a direct TLL() operation, p's if-condition in line 176 of L evaluates to `false`, and so p reads $(\perp,\perp,\perp)$ from $H[i][p]$ at $t_{L@175}$. This is a contradiction, and so Part (a) follows.

Now, we prove (7.34). Note that any process other than p can only change the value stored in $H[i][p]$ from $(\perp,\perp,\perp)$ to a non-$\perp$ value, and process p can write $(\perp,\perp,\perp)$ to $H[i][p]$ only during a `TLL(i)` operation. Process p does not write $(\perp,\perp,\perp)$ to $H[i][p]$ during $[t_{L@173}, t_{L@rsp}]$. By (7.33), S' completes during $[t_{L@173}, t_{L@175}]$. Thus to prove (7.34), it suffices to show that

$$H[i][p] \text{ has some non-}\perp \text{ value at some point during } S'. \tag{7.35}$$

If p' writes to $H[i][p]$ during S', then it must write a non-$\perp$ value to $H[i]$, and (7.35) follows. Hence, suppose process p' does not write to $H[i][p]$ during S'. As S' is a successful `TSC(i,·)`, either the if-condition in line 162 evaluates to `false` or the it evaluates to `true` but the $H[i][p]$.`SC()` operation in line 163 fails. In the case that the if-condition evaluates to `false`, $H[i][p]$ already stores some value $(x,g,q) \neq (\perp,\perp,\perp)$ when p' reads it at $t_{S'@157}$. Otherwise, p''s $H[i][p]$.`SC()` operation fails because another process, q, successfully executes an `SC()` operation on $H[i][p]$ in $(t_{S'@157}, t_{S'@163}) \subseteq [t_{L@173}, t_{L@175}]$ (by 7.33), and so $q \neq p$. Thus, q writes some non-$\perp$ value to $H[i][p]$ at some point during S'. Hence, (7.35) follows.

Proof of Part (a) if T is of type TRA. Suppose for the sake of contradiction that p' completes at least n `TWrite(i,·)` operations during $[t_{L@239}, t_{L@242}]$. Since p' increments its local variable $hCtr_{p'}[i]$ modulo n during each `TWrite(i,·)` operation, $hCtr_{p'}[i] = p$ at the invocation of at least one of these `TWrite(i,·)` operations. Let W' be the first such `TWrite(i,·)` operation. Thus,

$$[t_{W'@inv}, t_{W'@rsp}] \subseteq [t_{L@239}, t_{L@242}]. \tag{7.36}$$

So by (7.31) we know that

$$p' \text{ reads } c^* \text{ from } ReadCtr_p[i] \text{ in line 218 of } W'. \tag{7.37}$$

In the following, first we show that there exist x, g, and b, such that

$$H[i][p'][p] = H'[i][p'][p] = (x,g,c^*,b) \text{ at } t_{W'@rsp}. \tag{7.38}$$

Suppose process p' writes to neither $H[i][p'][p]$ nor $H'[i][p'][p]$ during W'. This implies that the if-condition in line 224 of W' evaluates to `false`. Since by (7.37), p' reads c^* from $ReadCtr_p[i]$ in line 218 of W', process p' reads (x,g,c^*,b) from both, $H[i][p'][p]$ and $H'[i][p'][p]$, in lines 220 and 221, respectively, for some x, g, and b. By the case assumption, p' does not write to $H[i][p'][p]$ and $H'[i][p'][p]$ during W', and by (7.32) p does not write to these variables during W'. Thus, $H[i][p'][p] = H'[i][p'][p] = (x,g,c^*,b)$ at $t_{W'@rsp}$.

Now suppose that p' writes to $H[i][p'][p]$ and $H'[i][p'][p]$ during W'. By (7.37), p' reads c^* from $ReadCtr_p[i]$ in line 218 and in line 226. Therefore, p' writes (x,g,c^*,b) to both $H[i][p'][p]$ and $H'[i][p'][p]$ in lines 225–228 of W', for some x, g, and b. Process p' does not write to $H[i][p'][p]$ during $(t_{W'@225}, t_{W'@rsp}]$, and not to $H'[i][p'][p]$ during $(t_{W'@228}, t_{W'@rsp}]$. By (7.32), process p also does not write to these variables during W', thus we have $H[i][p'][p] = H'[i][p'][p] = (x,g,c^*,b)$ at $t_{W'@rsp}$. Therefore, (7.38) follows.

After W', process p' does not write to $H[i][p'][p]$ and $H'[i][p'][p]$ as long as $H[i][p'][p] = H'[i][p'][p] = (x,g,c^*,b)$ and $ReadCtr_p[i] = c^*$. Other than process p', only process p can write to $H[i][p'][p]$ and $H'[i][p'][p]$. By (7.32), p does not write to those registers during $[t_{L@239}, t_{L@243}]$. By (7.31), $ReadCtr_p[i] = c^*$ throughout that interval. Thus, neither p' nor p writes to $H[i][p'][p]$ and $H'[i][p'][p]$ in $(t_{W'@rsp}, t_{L@243}]$. Therefore by (7.38), $H[i][p'][p] = H'[i][p'][p] = (x,g,c^*,b)$ throughout $[t_{W'@rsp}, t_{L@243}]$ and therefore throughout $[t_{L@242}, t_{L@243}]$, because according to (7.36), $t_{W'@rsp} \leq t_{L@242}$. This implies that p' reads (x,g,c^*,b) from $H[i][p'][p]$ and $H'[i][p'][p]$ in line 242 respectively line 243 of L, and hence the if-condition in line 244 evaluates to `true`. This contradicts the assumption that L is direct.

Proof of Part (b). We discussed at the beginning of this proof that process p' writes $[\![(x^*,g^*)]\!]\langle(x^*,g^*,p')\rangle$ to $A[i]$ $[\![$in line 158$]\!]\langle$either in line 227 or in line 232$\rangle$ of S at at some point t, such that $[\![t < t_{L@173}]\!]\langle t < t_{L@239}\rangle$, and no process writes to $A[i]$ during $[\![(t, t_{L@173}]]\!]\langle(t, t_{L@239}]\rangle$. Thus, at $[\![t_{L@173}]\!]\langle t_{L@239}\rangle$ process p' has not completed a $[\![$successful `TSC(i,·)`$]\!]\langle$`TWrite(i,·)`$\rangle$ operation following S, as otherwise the value stored in $A[i]$ would

change $[\![$in line 158$]\!]\langle$either in line 227 or in line 232$\rangle$. By Part (a), p' completes fewer than n $[\![$successful $\texttt{TSC}(i,\cdot)]\!]\langle\texttt{TWrite}(i,\cdot)\rangle$ operations during $[\![[t_{L@173},t_{L@175}]]\!]\langle[t_{L@239},t_{L@242}]\rangle$. Thus, p' completes at most n $[\![$successful $\texttt{TSC}(i,\cdot)]\!]\langle\texttt{TWrite}(i,\cdot)\rangle$ operations during $[\![[t_{S@rsp},t_{L@175}]]\!]\langle[t_{S@rsp},t_{L@242}]\rangle$. Let t' be the point when p' finishes its $n+1$-th $[\![$successful $\texttt{TSC}(i,\cdot)]\!]\langle\texttt{TWrite}(i,\cdot)\rangle$ operation after S, and $t'=\infty$ if p' executes fewer than $n+1$ $[\![$successful $\texttt{TSC}(i,\cdot)]\!]\langle\texttt{TWrite}(i,\cdot)\rangle$ operations after S. Thus, we have $[\![t' > t_{L@175}]\!]\langle t' > t_{L@242}\rangle$.

As p' writes $[\![(x^*,g^*)]\!]\langle(x^*,g^*,p')\rangle$ to $A[i]$ during S, it also executes $\texttt{reserve}(g^*)$ in $[\![$line 156$]\!]\langle$line 219$\rangle$ of S. Thus, by Claim 7.9, $Rsrv(g^*,p',i) > 0$ throughout $[\![[t_{S@156},t']]\!]\langle[t_{S@219},t')\rangle$. As $[\![t_{S@158} < t_{L@173}$ and $t_{L@175} < t']\!]\langle t_{S@219} < t < t_{L@239}$ and $t_{L@242} < t'\rangle$, $Rsrv(g^*,p',i) > 0$ throughout $[\![[t_{L@173},t_{L@175}]]\!]\langle[t_{L@239},t_{L@242}]\rangle$.

Proof of Part (c). Process p reads $[\![(x^*,g^*)]\!]\langle(x^*,g^*,p')\rangle$ from $A[i]$ in $[\![$line 173$]\!]\langle$line 239$\rangle$ of L, thus it executes $\texttt{announce}(g^*)$ in $[\![$line 174$]\!]\langle$line 241$\rangle$ of L. Moreover, since L is a direct operation, the if-condition in $[\![$line 176$]\!]\langle$line 244$\rangle$ evaluates to false, and so p does not call $\texttt{unannounce}(g^*)$ during L, and $\texttt{announce}(g^*)$ at not point other than in $[\![$line 174$]\!]\langle$line 241$\rangle$. Thus, after this point $Annc(g^*,p)$ is one larger than at $t_{L@inv}$ until L responds, and so (c) follows. □

7.5.5 Announced vs. Protected

Let p be a process and g^* a tag. Claim 7.16 states that number of times p has tag g^* protected is at least $Annc(g^*,p)$, unless p is at some specific points in the execution. Claim 7.17 says that for a completed operation M, $Annc(g^*,p)$ is the same right before and right after M, if M is not a $[\![\texttt{TLL()}]\!]\langle\texttt{TRead()}\rangle$ operation by p with return value g^*, a $\texttt{Protect}(g^*)$, an $\texttt{Unprotect}(g^*)$, or a $\texttt{CancelProtect}(g^*)$. Using this together with Claim 7.14 and Claim 7.15, we show that if at some point a process has completed in total more $\texttt{Protect}(g^*)$ and $[\![\texttt{TLL()}]\!]\langle\texttt{TRead()}\rangle$ operations with return value $(\cdot,g^*)$ than $\texttt{Unprotect}(g^*)$ calls, then $Annc(g^*,p) > 0$ at that point (see Claim 7.18). Invariant 7.19 is that $Annc(g^*,p)$ is always at least 0. Finally, Claim 7.20

states the relation between protected, announced, and reserved.

Claim 7.16. *For tag g^*, any process p, and any point t, one of the following is true.*

(a) $Protected(g^*,p) \geq Annc(g^*,p)$, *or*

(b) $Protected(g^*,p) \geq Annc(g^*,p) - 1$, *and either p is poised to execute ⟦line 175 or 178 of a* `TLL(·)`*⟧⟨line 242, 243 or 246 of a* `TRead(·)`*⟩ operation, where p's local variable g has value g^*, or p's last step on the ⟦TLSA⟧⟨TRA⟩ object was a* `Protect(`g^*`)` *call that did not succeed at or before t.*

Proof. Initially, no tag is protected or announced by p, i.e., $Protected(g^*,p) = 0$ and $Annc(g^*,p) = 0$. So Part (a) at $t = 0$.

Now suppose that there is a point t, such that the claim holds throughout $[0,t)$. In the rest of this proof, for any point t^*, let $P_{\prec t^*}$, respectively $P_{\succ t^*}$, denote the value of $Protected(g^*,p)$ immediately before, respectively immediately after, t^*. Similarly, let $A_{\prec t^*}$, respectively $A_{\succ t^*}$, denote the value of $Annc(g^*,p)$ immediately before, respectively immediately after, t^*.

Case 1. First suppose that Part (a) holds immediately before t, and at t, either $Annc(g^*,p)$ increases, or $Protected(g^*,p)$ decreases. The former can happen only if p executes `announce(`g^*`)` in ⟦line 174 or 182 of a `TLL()`⟧⟨line 241 or 247 of a `TRead()`⟩ operation, or line 206 of a `Protect(`g^*`)` (this will be covered by Cases 1.1-1.3 below). The latter happens if at t, process p executes `Unprotect(`g^*`)`, or a `CancelProtect(`g^*`)` after p's last `Protect(`g^*`)` call succeeded (Case 1.4 below).

Case 1.1 Consider the case in which p executes ⟦line 174 of a `TLL()`⟧⟨line 241 of a `TRead()`⟩ operation L at t. As p only executes `announce(`g^*`)` once between the invocation and the point immediately after t, we have $A_{\succ t} = A_{\prec t_{L@inv}} + 1$. The claim holds throughout $[0,t)$, and so at $t_{L@inv}$. Process p is not poised to execute ⟦line 175 or 178⟧⟨line 242, 243 or 246⟩ at the invocation of L. Moreover, since we are considering a good transcript, by property (G4), if p's

last operation call before L is a `Protect`(g^*) call, then that call must have succeeded before L is invoked (as otherwise p would have to call a `CancelProtect()` before L). Thus, Part (a) holds at $t_{L@inv}$. I.e., $P_{\prec t_{L@inv}} \geq A_{\prec t_{L@inv}}$. Process p does not call `Unprotect`(g) during $[t_{L@inv}, t]$, thus, $P_{\succ t} \geq P_{\prec t_{L@inv}}$. Therefore, $P_{\succ t} \geq P_{\prec t_{L@inv}} \geq A_{\prec t_{L@inv}} = A_{\succ t} - 1$. Moreover, p is poised to execute ⟦line 175⟧⟨line 242⟩ immediately after t. Thus, Part (b) is true, and so the claim holds for this case.

Case 1.2. Now suppose that process p executes ⟦line 182 of a `TLL()`⟧⟨line 247 of a `TRead()`⟩ operation L at t. For some tag g, process p executes `announce`(g), `unannounce`(g) and `announce`(g^*) in ⟦lines 174, 178 and 182⟧⟨lines 241, 246 and 247⟩ of L respectively. Thus, $A_{\succ t} = A_{\prec t_{L@inv}} + 1$. With the same argument as in Case 1.1, $P_{\prec t_{L@inv}} \geq A_{\prec t_{L@inv}}$. Operation L is an indirect ⟦`TLL()`⟧⟨`TRead()`⟩ call with return value $(\cdot, g^*)$, and it linearizes at some point during ⟦$[t_{L@inv}, t_{L@175}]$⟧⟨$[t_{L@238}, t_{L@243}]$⟩ (by def of $lin()$ and Claim 7.14(b)) ⟦or at $t_{L@179}$⟧. Thus, $Protected(g^*, p)$ increases at some point during $[t_{L@inv}, t)$, i.e. $P_{\succ t} = P_{\prec t_{L@inv}} + 1$. Hence, $P_{\succ t} = P_{\prec t_{L@inv}} + 1 \geq A_{\prec t_{L@inv}} + 1 = A_{\succ t}$, and so the claim holds for this case.

Case 1.3. Suppose that process p executes line 206 of a `Protect`(g^*) operation P at t. Then p also executes `announce`(g^*) at t, so $A_{\succ t} = A_{\prec t} + 1$. Since it is assumed that Part (a) holds immediately before t, we have $P_{\prec t} \geq A_{\prec t}$. If P succeeds at t, then $Protected(g^*, p)$ increases by 1 at t, and hence $P_{\succ t} = P_{\prec t} + 1 \geq A_{\prec t} + 1 = A_{\succ t}$, and so Part (a) holds immediately after t. If P does not succeed at t, then $P_{\succ t} = P_{\prec t} \geq A_{\prec t} = A_{\succ t} - 1$. But since p just finished a `Protect`(g^*) call that did not succeed before t, then Part (b) holds immediately after t.

Case 1.4. Next suppose that $Protected(g^*, p)$ decreases. This implies that at t, process p executes `Unprotect`(g^*) or a `CancelProtect`(g^*) that is called because the process did not know that its last `Protect`(g^*) call succeeded. Hence, we have $P_{\succ t} = P_{\prec t} - 1$ and $P_{\prec t} \geq A_{\prec t}$. Note that `Unprotect`$(g^*)$ and `CancelProtect`(g^*) both consist of only one shared memory step and that is writing to an entry of Emp during `unannounce`(g^*). Thus, immediately after t

we have $A_{\succ t} = A_{\prec t} - 1$. Therefore, $P_{\succ t} = P_{\prec t} - 1 \geq A_{\prec t} - 1 = A_{\succ t}$, and so the claim holds.

Case 2. Now suppose that Part (b) holds immediately before t, and at t process p executes ⟦line 175 or 178⟧⟨line 242, 243 or 246⟩ (this is covered in Cases 2.1-2.2 below), or p takes one step while p's last `Protect(`g^*`)` call did not succeed at or before t (Case 2.3 below), or the value of $Protected(g^*, p)$ or $Annc(g^*, p)$ changes, such that immediately after t, p is still poised to execute ⟦line 175 or 178⟧⟨line 242, 243 or 246⟩, or p's last step on the ⟦TLSA⟧⟨TRA⟩ object was a `Protect(`g^*`)` call that did not succeed at or before t (Case 2.4 below).

Case 2.1. Suppose that at t, process p executes ⟦line 175 of a `TLL()`⟧⟨line 242 or line 243 of a `TRead()`⟩ operation L, where p's local variable g has value g^*.

First suppose L is a direct ⟦`TLL()`⟧⟨`TRead()`⟩ operation. Process p executes `announce(`g^*`)` exactly once during $[t_{L@inv}, t]$, thus, $A_{\succ t} = A_{\prec t_{L@inv}} + 1$. This operation linearizes at ⟦$t_{L@173}$⟧⟨$t_{L@239}$⟩ which is during $[t_{L@inv}, t]$, so $P_{\succ t} = P_{\prec t_{L@inv}} + 1$. The claim holds throughout $[0, t)$, and so at the invocation of L. With the same argument as in Case 1.1, $P_{\prec t_{L@inv}} \geq A_{\prec t_{L@inv}}$. Therefore, $P_{\succ t} = P_{\prec t_{L@inv}} + 1 \geq A_{\prec t_{L@inv}} + 1 = A_{\succ t}$, and so the claim is true for this case.

Now suppose L is an indirect ⟦`TLL()`⟧⟨`TRead()`⟩ operation. By executing ⟦line 175⟧⟨line 242 or line 243⟩, $Annc(g^*, p)$ does not change, so $A_{\succ t} = A_{\prec t}$. Operation L may linearize with return value $(\cdot, g^*)$ at t (see Claim 7.14), thus $P_{\succ t} \geq P_{\prec t}$. Since it is assumed that Part (b) holds immediately before t, $P_{\prec t} \geq A_{\prec t} - 1$. Thus, $P_{\succ t} \geq P_{\prec t} \geq A_{\prec t} - 1 = A_{\succ t} - 1$. Moreover, as L is an indirect operation, p's next shared memory step immediately after t is executing ⟦line 178⟧⟨line 242 respectively line 246⟩. Thus, Part (b) is true, and so the claim holds for this case.

Case 2.2. Suppose that at t, process p executes ⟦line 178 of a `TLL()`⟧⟨line 246 of a `TRead()`⟩ operation L, where p's local variable g has value g^*. Therefore, $A_{\succ t} = A_{\prec t} - 1$. Operation L linearizes at some point during ⟦$[t_{L@inv}, t_{L@175}]$⟧⟨$[t_{L@238}, t_{L@243}]$⟩ (by def of $lin()$ and Claim 7.14(b)) ⟦or at $t_{L@179}$⟧. Thus, the number of times tag g^* is protected by p does not

change at $[\![t = t_{L@178}]\!]\langle t = t_{L@246}\rangle$, which means $P_{\succ t} = P_{\prec t}$. Since it is assumed that Part (b) holds immediately before t, $P_{\prec t} \geq A_{\prec t} - 1$, therefore, $P_{\succ t} = P_{\prec t} \geq A_{\prec t} - 1 = A_{\succ t}$, and so Part (a) is true immediately after t, hence the claim holds at this point.

Case 2.3. Next suppose that p's last step on the $[\![$TLSA$]\!]\langle$TRA$\rangle$ object is a `Protect`(g^*) call that does not succeed at or before t, and at t, process p takes one step on the $[\![$TLSA$]\!]\langle$TRA$\rangle$ object. As we are considering only good transcripts, by property (G4), process p must execute `CancelProtect`(g^*) call C at t. This operation consists of only one shared memory step and that is calling `unannounce`(g^*). Hence $A_{\succ t} = A_{\prec t} - 1$. Since p's last `Protect`(g^*) call before t does not succeed and so does not increment $Protected(g^*, p)$, C also does not change the value of $Protected(g^*, p)$, and so we have $P_{\succ t} = P_{\prec t}$. By the case assumption, $P_{\prec t} \geq A_{\prec t} - 1$, so we have $P_{\succ t} = P_{\prec t} \geq A_{\prec t} - 1 = A_{\succ t}$, and therefore Part (a) is true immediately after t.

Case 2.4. Suppose that at t, the value of $Protected(g^*, p)$ or $Annc(g^*, p)$ changes, and immediately after t the following holds.

Either p is poised to execute one of $[\![$lines 175 and 178$]\!]\langle$lines 242, 243 and 246$\rangle$.
where p's local variable g has value g^*, or p's last step before t on the $[\![$TLSA$]\!]\langle$TRA$\rangle$ object is a `Protect`(g^*) call that does not succeed at or before t. (7.39)

Since Part (b) holds immediately before t, (7.39) also holds immediately before t.

The value of $Protected(g^*, p)$ or $Annc(g^*, p)$ can only change if p takes a shared memory step on the $[\![$TLSA$]\!]\langle$TRA$\rangle$ object. If p executes $[\![$line 175$]\!]\langle$line 242 or 243$\rangle$ at t, then p is poised to execute $[\![$line 178$]\!]\langle$line 243 respectively 246$\rangle$ immediately after t. However, this step does not change the value of $Protected(g^*, p)$ or $Annc(g^*, p)$. If p executes $[\![$line 178$]\!]\langle$line 246$\rangle$, then (7.39) does not hold immediately after t. If p's last step on the $[\![$TLSA$]\!]\langle$TRA$\rangle$ object before t is a `Protect`(g^*) call that did not succeed at or before t, then p'step on this object at t must be a `CancelProtect`(g^*) operation, because of property (G4) of a good transcript. Therefore, (7.39) does not hold immediately after t. Hence, this case cannot happen. □

Claim 7.17. *Consider some tag g^* and some process p. Let* OP *represent the set of all* Protect(g^*), Unprotect(g^*), CancelProtect(g^*) *and* ⟦TLL(·)⟧⟨TRead(·)⟩ *operations with return value $(\cdot, g^*)$, all executed by p in Λ. For an operation $M \notin$ OP, if $Annc(g^*, p) = k$ just before the invocation of M, then $Annc(g^*, p) = k$ at the response of M.*

Proof. The value of $Annc(g^*, p)$ changes only when p calls announce(g^*) or unannounce(g^*). Process p calls announce(g^*) and unannounce(g^*) only during an operation in OP, or during a ⟦TLL(·)⟧⟨TRead(·)⟩ operation with some return value $(\cdot, g') \neq (\cdot, g^*)$. Since $M \notin$ OP, the value of $Annc(g^*, p)$ can only change during M, if M is a ⟦TLL(·)⟧⟨TRead(·)⟩ operation that does not return $(\cdot, g^*)$.

Suppose M is a direct ⟦TLL(·)⟧⟨TRead(·)⟩ operation, that does not return $(\cdot, g^*)$. The only way that p can change the value of $Annc(g^*, p)$ during M is if p calls announce(g^*) in ⟦line 174⟧⟨line 241⟩ of M. But p does not do so, because otherwise M would return $(\cdot, g^*)$.

Next suppose that M is an indirect ⟦TLL(·)⟧⟨TRead(·)⟩ operation that returns some value $(x', g') \neq (\cdot, g^*)$ which it reads from ⟦$H[i][p]$ in line 175⟧⟨both $H[i][p][q]$ and $H'[i][p][q]$, for some q, in lines 242 and 243⟩. Since $g' \neq g^*$, p's execution of announce(g') in ⟦line 182⟧⟨line 247⟩ of M does not change the value of $Annc(g^*, p)$. Moreover, if process p calls announce(g^*) in ⟦line 174⟧⟨line 241⟩ of M, then and only then it executes unannounce(g^*) in ⟦line 178⟧⟨line 246⟩ of M, because M is an indirect ⟦TLL(·)⟧⟨TRead(·)⟩ operation. Thus, $Annc(g^*, p) = k$ at the response of M. □

Claim 7.18. *Let t be a point in time at which p has no pending operation call on the ⟦TLSA⟧⟨TRA⟩ object. Then $Protected(g, p) = Annc(g, p) > 0$ at t.*

Proof. By Claim 7.14 and Claim 7.15, $Annc(g, p)$ is one larger at the response of any ⟦TLL()⟧⟨TRead()⟩ operation with return value $(\cdot, g)$ than at the invocation of this operation. Each Protect(g) by p increases the value of $Annc(g, p)$ by 1, and each Unprotect(g) and CancelProtect(g) decreases the value of $Annc(g, p)$ by 1. By Claim 7.17 the value of $Annc(g, p)$ is the same at the invocation and at the response of any other operation.

Moreover, any `Protect`(g) call by p that is not effective at t, is immediately followed the invocation of a `CancelProtect`(g) call in $\Lambda|p$ before t. Since no operation is pending at t, the number of `CancelProtect`(g) calls by p that are completed before t is the same as the number of `Protect`(g) calls by p that are completed before t and are not effective at t.

Thus, the value of $Annc(g,p)$ at t is the total number of operation calls M by p that are completed before t, where M is either a $[\![$`TLL()`$]\!]\langle$`TRead()`$\rangle$ with return value $(\cdot,g)$, or a `Protect`(g) call that is effective at t, minus the number of `Unprotect`(g) calls by p that are completed before t. Thus, since no operation by p is pending at t,

$$Annc(g,p) = \alpha(p,g,t) + \theta(p,g,t) - \gamma(p,g,t) \text{ at } t, \tag{7.40}$$

which is the same as $Protected(g,p)$. □

Invariant 7.19. *At each point in time* $Annc(g^*,p) \geq 0$.

Proof. Suppose that for some point t, the invariant holds throughout $[0,t)$, and at t, some process takes one step. We show that the invariant holds right after t. Since only p change the value of $Annc(g^*,p)$, the invariant can only become false, if at point t process p executes an `unannounce`(g^*) call, and this call changes the value of $Annc(g^*,p)$ to a negative value. Process p calls `unannounce`(g^*) either in $[\![$line 178$]\!]\langle$line 246$\rangle$ of an indirect $[\![$`TLL()`$]\!]\langle$`TRead()`$\rangle$ operation, or in line 207 of an `Unprotect`(g^*) operation, or in line 208 of an `CancelProtect`(g^*) operation.

First consider the case that at t, process p executes `unannounce`(g^*) in $[\![$line 178$]\!]\langle$line 246$\rangle$ of an indirect $[\![$`TLL()`$]\!]\langle$`TRead()`$\rangle$ operation L. By the implementation, p executes an `announce`(g^*) operation in $[\![$line 174$]\!]\langle$line 241$\rangle$ of L, and since we assumed $Annc(g^*,p) \geq 0$ at any point before t, we have $Annc(g^*,p) > 0$ right after p executes $[\![$line 174$]\!]\langle$line 241$\rangle$ of L. The value of $Annc(g^*,p)$ does not change after this point until p executes `unannounce`(g^*) in $[\![$line 178$]\!]\langle$line 246$\rangle$ of L at t. Thus, we have $Annc(g^*,p) \geq 0$ right after t.

Next consider the case that at t, process p executes `unannounce`(g^*) in line 207 of an `Unprotect`(g^*) call U. Since Λ is a good transcript, process p protects tag g right before the

invocation of U. Operation call U consists of only one shared memory step, and so p protects tag g right before t. Thus, by Claim 7.18, $Annc(g^*, p) > 0$ just before t, and so $Annc(g^*, p) \geq 0$ right after t.

Finally consider the case that at t, process p executes `unannounce`(g^*) in line 208 of a `CancelProtect`(g^*) operation C. In a good transcript, a process p can only call `CancelProtect`(g^*) after a `Protect`(g^*) call and before it executes any other operation on the $[\![$TLSA$]\!]\langle$TRA$\rangle$ object. Let P be p's last `Protect`(g^*) call before C. As $Annc(g^*, p) \geq 0$ at any point before t, and therefore right before operation P is executed, we have $Annc(g^*, p) > 0$ right after P. Process p does not execute any operation on the $[\![$TLSA$]\!]\langle$TRA$\rangle$ object after P and before C. Thus, $Annc(g^*, p) > 0$ right before C is executed and so right before t. Hence, we have $Annc(g^*, p) \geq 0$ right after t. □

Claim 7.20. *For any tag g^*, any process p, and any point t, if at t, $Protected(g^*, p) > 0$, then g^* is announced or reserved at t.*

Proof. If at point t, process p has no pending operation on the $[\![$TLSA$]\!]\langle$TRA$\rangle$ object, then by Claim 7.18, $Annc(g^*, p) > 0$ at t. Now suppose p has a pending operation call M at point t, and g is protected at t. Note that each of `Protect()`, `Unprotect()`, and `CancelProtect()` operations consists of one shared memory step, and so p's execution cannot be pending during any of these operations.

Case 1. First consider the case in which tag g is not protected by p at the invocation of M. Since g^* is protected at t, and since M is not a `Protect`(g^*) call, then M must be a $[\![$`TLL()`$]\!]\langle$`TRead()`$\rangle$ operation with return value $(\cdot, g^*)$, and $lin(M) < t$.

Case 1.1. Suppose M is a direct $[\![$`TLL()`$]\!]\langle$`TRead()`$\rangle$ operation with return value $(\cdot, g^*)$. Therefore, $[\![lin(M) = t_{M@173}]\!]\langle lin(M) = t_{M@239}\rangle$. By Claim 7.15(b), some process p has tag g^* reserved throughout $[\![[t_{M@173}, t_{M@175}]]\!]\langle[t_{M@239}, t_{M@242}]\rangle$. By Claim 7.15(c), $Annc(g^*, q)$ is one larger at any point during $[\![[t_{M@174}, t_{M@rsp}]]\!]\langle[t_{M@241}, t_{M@rsp}]\rangle$ than at $t_{M@inv}$. By Invariant 7.19,

$Annc(g^*,q)$ is always at least 0 and so at the invocation of M. Hence, we have $Annc(g^*,q) > 0$ throughout $[\![[t_{M@174}, t_{M@rsp}]]\!]\langle[t_{M@241}, t_{M@rsp}]\rangle$. Therefore the claim follows for this case.

Case 1.2. Next suppose M is an indirect $[\![\texttt{TLL()}]\!]\langle\texttt{TRead()}\rangle$ operation. By Claim 7.14, there exists a $[\![\texttt{TSC()}]\!]\langle\texttt{TWrite()}\rangle$ operation S by some process p such that p has tag g^* reserved throughout $[\![[t_{S@158}, t_{M@rsp}]]\!]\langle[t_{W@227}, t_{R@rsp}]\rangle$. Moreover, in this case $lin(M)$ is $[\![$either $t_{S@158}$ or $t_{M@179}]\!]\langle t_{S@227}\rangle$. $[\![$By the same claim we know that $t_{S@158} \leq t_{M@175}$, therefore, $lin(M) \geq t_{S@158}]\!]$. Hence, tag g^* is reserved throughout $[lin(M), t_{M@rsp}]$, and so at t.

Case 2. Now consider the case in which tag g is protected at the invocation of M. We show that $Annc(g^*,p) > 0$ throughout $[t_{M@inv}, t_{M@inv}]$, and so at t, which proves the claim for this case. By Claim 7.18, $Annc(g^*,p) > 0$ just before the invocation of M. So $Annc(g^*,p)$ can only become 0, if p executes an `unannounce`(g^*) call during M. Since M is not an `Unprotect()` or a `CancelProtect()` call, then M must be a $[\![\texttt{TLL()}]\!]\langle\texttt{TRead()}\rangle$ operation, in which p executes `unannounce`(g^*) in $[\![$line 178$]\!]\langle$line 246$\rangle$. However, this process must have executed `announce`(g^*) in $[\![$line 174$]\!]\langle$line 241$\rangle$ of M, and so $Annc(g^*,p) > 1$ just before p executes $[\![$line 178$]\!]\langle$line 246$\rangle$ during M. Thus, $Annc(g^*,p) > 0$ throughout $[t_{M@inv}, t_{M@inv}]$. □

7.5.6 What Do Emp and Use Represent?

The following invariant describes the relation between Emp, $Annc$, $Rsrv$. Particularly, it shows that for any process q and any block b_x, we have $Emp_q[x] > 0$ if and only if there is a process q and a tag $g \in b_x$, such that $Annc(g,q) > 0$ or $Rsrv(g,q,i) > 0$ for some $i \in \{0,\ldots,m-1\}$.

Invariant 7.21. *For every block b_x and any process q the following holds.*

$$Emp_q[x] = \sum_{i=0}^{m-1} \sum_{g \in b_x} Rsrv(g,q,i) + \sum_{g \in b_x} Annc(g,q).$$

Proof. First we show that the statement holds at the beginning of the transcript at time 0. Initially no process has any tag announced or reserved, and so both $Rsrv(g,q,i)$ and $Annc(g,q)$

have value 0, for any tag g, and any object $A[i]$. Moreover, $Emp_q[x] = 0$, for all blocks b_x. Thus, the invariant holds at 0.

Now suppose that the invariant holds throughout $[0,t)$, for some t. We show that if at t the value of $Emp_q[x]$, $Annc(g,q)$, or $Rsrv(g,q,i)$ changes, for some $g \in b_x$ and $i \in \{0,\ldots,m-1\}$, then the invariant holds right after that point. Only process q can write to $Emp_q[x]$, and it does so only during its `reserve(`g`)` (line 210), `unreserve(`g`)` (line 212), `announce(`g`)` (line 214), and `unannounce(`g`)` (line 216), for some $g \in b_x$. Each of these operations consists of one shared memory step in which $Emp_q[x]$ gets modified. Thus by definition of $Annc(g,q)$ and $Rsrv(g,q,i)$, whenever the value of $Emp_q[x]$ is modified, the value of either $Annc(g,q)$ or $Rsrv(g,q,i)$ gets modified as well, for some $g \in b_x$ and $i \in \{0,\ldots,m-1\}$. Thus if at point t, process q increments $Emp_q[x]$ in line 210 of a `reserve(`g`)` or line 214 of an `announce(`g`)`, then and only then $Rsrv(g,q,i)$ respectively $Annc(g,q)$ are incremented as well. Similarly, if at t, process q decrements $Emp_q[x]$ in line 212 of a `unreserve(`g`)` or line 216 of an `unannounce(`g`)`, then and only then $Rsrv(g,q,i)$ respectively $Annc(g,q)$ are decremented. □

The next invariant describes how Act an $NumOfActive$ are related. Moreover, it shows that $\sigma(p,x)$ has value 0 if p is not during a cleaning interval of b_x.

Invariant 7.22. *For every block b_x owned by some process p the following holds.*

$$\sigma(p,x) + \sum_{q=0}^{n-1} Act_q[x] = NumOfActive(x), \text{ and}$$

$$\sigma(p,x) = 0 \text{ at any point } t \text{ that is not during a cleaning interval of } b_x.$$

We defer the proof of this invariant to the end of this section (page 242). Most of the following claims prove some properties of the transcript assuming that Invariant 7.22 holds during a fixed prefix of the transcript. Later, in order to prove that this invariant holds throughout the transcript, we show that it holds at the beginning of any transcript, and if the invariant holds at any prefix of the transcript, then it also holds right after a process takes one more shared memory step. To

do so, we can use the properties proved in the following claims given the invariant holds during the prefix.

7.5.7 GetFree() Returns a Free Tag

In this section, we assume in each claim that Invariant 7.22 holds in a clearly specified prefix of Λ. Having that in mind, we first (in Claim 7.23) show that when a block is free (see (7.7) on page 194), then all tags of that block are free. In Claim 7.24, we argue that once a tag is free, it remains free until it is returned from a `GetFree()` operation. Observation 7.26 states the properties of an ABA-detecting register. Claim 7.25 and Claim 7.27 discuss about what we know before and after a search interval. In Claim 7.28 and Claim 7.29, we show that all tags of a block b_y are free starting from the point when a cleaning interval of b_y starts and until the first tag of this block is returned from a `GetFree()` operation. In Corollary 7.30, we mention an immediate result of Claim 7.29 which will help us later to prove Invariant 7.22. Finally Claim 7.31 puts all these results together to show that a tag that is returned by a `GetFree()` operation is free right before this `GetFree()` operation linearizes.

Claim 7.23. *Let b_x be a block that p owns and t be a point in time, such that Invariant 7.22 holds throughout $[0,t]$. If b_x is free at t, then*

(a) no tag of b_x is active, announced, or reserved at t, and

(b) all tags of b_x are free at t.

Proof. Recall that a block b_x is free if $\sigma(p,x) + \sum_{q=0}^{n-1} Act_q[x] = 0$, and $\forall q \in \{0,\ldots,n-1\}$: $Emp_q[x] = 0$. Thus by Invariants 7.8–7.21, no tag of b_x is announced or reserved at this point. Moreover, by Invariant 7.22 no tag of b_x is active at t, and so Part ((a)) follows. By Claim 7.20 and Corollary 7.10, if a tag is not reserved or announced by any process, then it is not protected or stored in any element of A. Thus, no tag of b_x is active, protected, or stored in an element of A at t, and so this proves Part ((b)). □

In the following, we show that once a tag becomes free, it remains free until the process which owns the tag returns it from a `GetFree()`.

Claim 7.24. *Let g^* be a tag that some process p owns. Suppose g^* is free at some point t_1, and let t_2 be the first point after t_1 at which a* `GetFree()` *by p returns g^*, and $t_2 = \infty$ if p does not execute any following* `GetFree()` *which returns g^*. Then g^* is free throughout $[t_1, t_2)$.*

Proof. Suppose for the sake of contradiction that at some point during $[t_1, t_2)$ tag g^* becomes occupied. Let t be the first such point. I.e. at point t, a process either makes g^* active, protects it, or writes it to an element of A. This implies that

$$g^* \text{ is free throughout } [t_1, t). \tag{7.41}$$

Case 1. First assume that at t, process p makes tag g^* active. I.e. at t, a `GetFree()` operation by p which returns g^* linearizes. By the definition of linearization points, that is when the `GetFree()` operation by p returns g^*. However, by the assumption, no `GetFree()` operation by p returns g^* throughout $[t_1, t_2)$ and therefore not at t. This is a contradiction.

Case 2. Next suppose that at t, some process q writes tag g^* to $A[i]$, for some $0 \le i < m$. This can be done only in $[\![$line 158 of a successful $\texttt{TSC}(i, (\cdot, g^*))]\!]\langle$line 227 or line 232 of a $\texttt{TWrite}(i, (\cdot, g^*))\rangle$ operation S by q. Note that for a $[\![$successful $\texttt{TSC()}]\!]\langle\texttt{TWrite()}\rangle$ operation S, we have $[\![lin(S) = t_{S@158}]\!]\langle lin(S) = t_{S@227}$ or $t_{S@232}\rangle$, thus $t = lin(S)$. Since we are considering only good transcripts, g^* must be occupied throughout $[t_{S@inv}, lin(S)] = [t_{S@inv}, t]$, which contradicts (7.41).

Case 3. Now suppose at t, tag g^* becomes protected, i.e. a $[\![\texttt{TLL}(i)]\!]\langle\texttt{TRead}(i)\rangle$ operation L by some process q with return value (x^*, g^*) linearizes at t, for some $0 \le i < m$ and some x^*. If L is a direct $[\![\texttt{TLL}(i)]\!]\langle\texttt{TRead}(i)\rangle$, then by definition $lin(L) = [\![t_{L@173}]\!]\langle t_{L@239}\rangle$, and q reads $[\![(x^*, g^*)]\!]\langle(x^*, g^*, \cdot)\rangle$ from $A[i]$ at $t = lin(L)$. Thus, $A[i] = [\![(x^*, g^*)]\!]\langle(x^*, g^*, \cdot)\rangle$ right before t, which contradicts (7.41). $[\![$If L is an indirect $\texttt{TLL}(i)$ and q's if-condition in line 180 evaluates to

false, then $lin(L) = t_{L@179}$ which is the point when q reads the return value of L from $A[i]$. Thus, $A[i] = (x^*, g^*)$ right before t, which contradicts (7.41).$]\!]$ If L is an indirect $[\![$TLL(i) and q's if-condition in line 180 evaluates to true$]\!]\langle$TRead$(i)\rangle$, then $lin(L)$ is the point at which some process writes $[\![(x^*, g^*)]\!]\langle(x^*, g^*, \cdot)\rangle$ to $A[i]$ during a $[\![$successful TSC$(i,(x^*,g^*))]\!]\langle$TWrite$(i,(x^*,g^*))\rangle$ (and by Claim 7.14 such a point exists). With the same arguments as in Case 2 this yields a contradiction as well.

Case 4. Finally, suppose that at t, a Protect(g^*) call by some process q succeeds. Since Λ is a good transcript, g^* is occupied at right before t. This contradicts (7.41). □

Next we show that just before some process p starts a search interval, its local variables sum_p and sum'_p are reset to 0 and false, respectively.

Claim 7.25. *For each process p, $sum_p = 0$ and $sum'_p =$ false at the beginning of each of its search interval.*

Proof. Consider some search interval by p that starts at some point t. We have $\rho_p = 0$ when a search interval starts in line 185 of a GetFree() operation. Thus, $\rho_p = 0$ at t. The value of p's local variable ρ_p can only change in two places; It gets incremented modulo $3n$ in line 189 of each GetFree() operation and it is reset in line 191 of a GetFree() operation during which the if-condition in line 190 evaluates to true.

Initially $sum_p = 0$ and $sum'_p =$ false. Thus, if t happens during the first GetFree() by p, then the claim holds. Now consider the last GetFree() operation G that p executes before t. First suppose that the value of ρ_p changes to 0 for the last time before t in line 191 of G. At the same point, p writes 0 to sum_p and false to sum'_p (line 191 of G). As neither sum_p nor sum'_p is modified between this point and t, the claim follows for this case.

Now suppose that the last time the value of ρ_p changes to 0 before t is in line 189 of G. This implies that p has completed $3n-1$ GetFree() operations before G during which it did not reset the value of ρ_p, but only incremented this value. Thus, there are $3n-1$ GetFree() operations

$G_1,\ldots,G_{3n-1}$ that p executes before G, such that in line 189 of G_i process p increments ρ_p to i, for $i \in \{0,\ldots,3n-1\}$. Consider G_{2n}, and recall that p does not reset ρ_p during this operation. Thus, the if-condition in line 190 evaluates to `false`. As $\rho_p = 2n$ right after p executes line 189 of G_{2n} at some point t', this implies that we have $sum_p = 0$ and $sum'_p =$ `false` when p checks the if-condition in line 190 of G_{2n} at t'. Now we prove that the value of sum_p and sum'_p do not change throughout $[t',t]$, and so the claim follows. This is because, these two variables can only get modified in line 187 if $0 \leq \rho_p < n$, in line 188 if $n \leq \rho_p < 2n$, and in line 191. However, none of these happens after t' during $G_{2n}, G_{2n+1},\ldots,G_{3n-1}, G$, and before t, and so the value of sum_p and sum'_p remains 0, respectively, `false` throughout $[t',t]$. □

This observation reminds us of the properties of an ABA-detecting register which is introduced and implemented in Chapter 5. This will be used later in Claim 7.27.

Observation 7.26. *Let X be a multi-reader multi-writer ABA-detecting register. If p executes two consecutive X.`DRead()` operations at t_1 and $t_2 > t_1$, such that it reads $(\cdot,$`false`$)$ at t_2, then no process executes X.`DWrite()` operations throughout $[t_1,t_2]$.*

Proof. This follows from the definition of an ABA-detecting register as defined in (Aghazadeh and Woelfel, 2015). □

Now using Observation 7.26, we show that depending on the value of sum_p and sum'_p at the end of a search interval I of some block b_x by p, we can decide whether b_x is free at some point during I.

Claim 7.27. *Let I be a search interval of some block b_x that p owns, and suppose Invariant 7.22 holds from point 0 until the end of I.*

(a) If $sum_p = 0$ and $sum'_p =$ `false` at the end of I, then block b_x is free at some point during this search interval, and

(b) if $sum_p \neq 0$ or $sum'_p \neq$ `false` at the end of I, then block b_x is not free at some point during this search interval.

Proof. During a search interval of b_x, process p reads each $Emp_0[x],\ldots,Emp_{n-1}[x]$ and $Act_0[x],\ldots,Act_{n-1}[x]$ twice. Let $(x_q,\cdot)$ and $(x'_q,\cdot)$ be the value p reads from $Emp_q[x]$ and $Act_q[x]$, respectively, in its first read, and $(\cdot,f_q)$ and $(\cdot,f'_q)$ be the value p reads from $Emp_q[x]$ and $Act_q[x]$, respectively, in its second read, for $q \in \{0,\ldots,n-1\}$. Recall that by Claim 7.25, we have $sum_p = 0$ and $sum'_p = \texttt{false}$ at the beginning of each search interval. Thus, by the implementation at the end of this search interval of b_x, we have $sum_p = \sum_{q=0}^{n-1}(x_q + x'_q)$, and $sum'_p = f_0 \vee f'_0 \vee f_1 \vee f'_1 \vee \cdots \vee f_{n-1} \vee f'_{n-1}$.

Note that by Invariants 7.8–7.21 we can conclude that

$$\text{at any point } Emp_q[x] \geq 0 \text{ for all } q \in \{0,\ldots,n-1\}. \tag{7.42}$$

Moreover, $NumOfActive(x)$ represents the number of tags of block b_x that are active at each point, which is always a non-negative value. By Invariant 7.22, we have $\sigma(p,x) = 0$ during I, and so by the same Invariant, we have

$$\sum_{q=0}^{n-1} Act_q[x] \geq 0 \text{ during } I. \tag{7.43}$$

Suppose $sum'_p = \texttt{false}$ at the end of I. By Observation 7.26, this implies that no process wrote to $Emp_q[x]$ (similarly to $Act_q[x]$) and so $Emp_q[x] = x_q$ (similarly $Act_q[x] = x'_q$) during the interval that starts when p reads $Emp_q[x]$ (respectively $Act_q[x]$) for the first time and ends when p reads it for the second time during I. Thus there is a point t during search interval I (more specifically, after reading all $Emp_0[x],\ldots,Emp_{n-1}[x]$ and $Act_0[x],\ldots,Act_{n-1}[x]$ for the first time and before reading them for the second time) at which $Emp_q[x] = x_q$ and $Act_q[x] = x'_q$ for any $q \in \{0,\ldots,n-1\}$.

Now suppose that in addition to $sum'_p = \texttt{false}$, we have $sum_p = 0$ at the end of I, i.e. $sum_p = \sum_{q=0}^{n-1}(x_q + x'_q) = 0$ at the end of this interval, thus at t we have $\sum_{q=0}^{n-1}(Emp_q[x] + Act_q[x]) = 0$. Hence by (7.42) and (7.43), we have $Emp_q[x] = 0$ and $\sum_{q=0}^{n-1} Act_q[x] = 0$ at t, for all $q \in \{0,\ldots,n-1\}$. By Invariant 7.22, $\sigma(p,x) = 0$ throughout I and so at t, hence we can conclude that b_x is free at this point. This completes the proof of Part (a).

To prove Part (b), first suppose $sum'_p = \texttt{false}$ but $sum_p \neq 0$ at the end of I, i.e. $sum_p = \sum_{q=0}^{n-1}(x_q + x'_q) \neq 0$. Thus at t, we have $\sum_{q=0}^{n-1}(Emp_q[x] + Act_q[x]) \neq 0$. This implies by (7.42) and (7.43) that either there exists a $q \in \{0,\ldots,n-1\}$, such that $Emp_q[x] \neq 0$, or $\sum_{q=0}^{n-1} Act_q[x] \neq 0$ at t. By Invariant 7.22, $\sigma(p,x) = 0$ throughout I and so at t. Hence, we can conclude that b_x is not free at this point.

Finally suppose that $sum'_p \neq \texttt{false}$ at the end of I. By Observation 7.26, there is a $q \in \{0,\ldots,n-1\}$, such that either the value of $Emp_q[x]$ or $Act_q[x]$ changes between the first and the second read of that value. If the value of $Emp_q[x]$ changes, then by (7.42) $Emp_q[x] > 0$ at some point during I, and so b_x is not free at some point during this interval. If the value of $Act_q[x]$ changes, then $\sum_{q=0}^{n-1} Act_q[x] \neq 0$ at some point during this interval. Since by Invariant 7.22, we have $\sigma(p,x) = 0$ throughout I, therefore b_x is not free at some point during this interval. □

Consider a search interval of some block b_y that is immediately followed by a cleaning interval of the same block, and let t be a point after the beginning of this cleaning interval. In the following, we prove that if no tag of b_y is returned from a `GetFree()` starting from the beginning of the search interval until t, then all tags of b_y are free throughout the interval that starts at the beginning of the cleaning interval and ends at t.

Claim 7.28. *Consider some block b_y owned by process p. Suppose some cleaning interval of b_y starts at t_f, and Invariant 7.22 holds throughout $[0,t_f]$. Let $t_s \leq t_f$ be the beginning of the last search interval of b_y before t_f, and t any point after t_f. If no* `GetFree()` *operation by p returns a tag in b_y during $[t_s,t]$, then all tags of b_y are free throughout $[t_f,t]$.*

Proof. At t_f, process p executes line 194 for $\rho_p = 2n$. Thus by the implementation, we have $sum_p = 0$ and $sum'_p = \texttt{false}$ at t_f. Note that t_f is the end of a search interval of b_y by p, thus by Claim 7.27, block b_y is free at some point $t' \in [t_s,t_f)$. Therefore, by Claim 7.23, all tags of b_y are free at t'. Finally, by Claim 7.24, all tags of b_y remain free throughout $[t',t]$, and hence throughout $[t_f,t]$. □

Next we show that when p's `GetFree()` returns the first tag of a block b_y at some point t_y, then process p must have executed a cleaning interval of b_y before t_y, such that all tags of b_y are free throughout the interval that starts at the beginning of this cleaning interval and ends at t_y.

Claim 7.29. *Let t_x be a point at which p returns the first tag of block b_x, and let t_y be the first point after t_x at which p returns the first tag of a different block $b_y \neq b_x$. Also, let t_f be the beginning of the cleaning interval of b_y that p starts during $[t_x, t_y)$ (if there is no cleaning interval during $[t_x, t_y)$, then $t_f = t_y$). Assume Invariant 7.22 holds throughout $[0, t_f]$.*

(a) There exists at least one block $b_w \neq b_x$ that p owns, such that b_w is free throughout $[t_x, t_y)$.

(b) If G is the `GetFree()` by p from which the first tag of b_y is returned at t_y, then p executes line 196 during G.

(c) $t_f < t_y$ (i.e. p starts a cleaning interval of b_y at $t_f \in [t_x, t_y)$).

(d) The cleaning interval that starts at t_f ends before t_y.

(e) All tags of b_y are free throughout $[t_f, t_y)$.

Proof. We prove each part separately.

Proof of Part (a). First we show that

$$\text{at most } mn(2n+5) + \tau + n \text{ of } p\text{'s blocks contain tags that are active, announced or reserved at } t_x. \tag{7.44}$$

Let s be the number of processes that are poised to execute ⟦lines 156–160 of a `TSC(i,·)`⟧⟨lines 219–221 of a `TWrite(i,·)`⟩ operation at t_x. Also let ℓ be the number of processes that are poised to execute ⟦line 175 or 178 of a `TLL()`⟧⟨line 242, 243 or 246 of a `TRead()`⟩ operation at t_x, or their last step on the ⟦TLSA⟧⟨TRA⟩ object was a `Protect()` call that has not succeeded before or at t_x

By Claim 7.16, if tag g^* is announced by some process q at some point t, and q is not poised to execute ⟦line 175 or 178 of a `TLL()`⟧⟨line 242, 243 or 246 of a `TRead()`⟩ operation, where q's local variable has value g^*, or q's last step on the ⟦TLSA⟧⟨TRA⟩ object was a `Protect`(g^*) call that has not succeeded before or at t, then g^* is protected by q. Thus the number of announced tags is equal to the number of protected tags plus ℓ. Recall that all processes together can have at most τ tags active or protected. Thus, at most $\tau + \ell$ tags can be active or announced by all processes together at t_x.

Moreover, we know by Claim 7.7 that if a tag is reserved by some process q and q is not poised to execute ⟦lines 156–160 of a `TSC`$(i,\cdot)$⟧⟨lines 219–221 of a `TWrite`$(i,\cdot)$⟩ operation, then a copy of the tag is stored in some queue $rsrvQ_q[i]$, for some $i \in \{0,\ldots,m-1\}$. Hence the number of reserved tags is bounded by the number of tags in all processes' queues, plus the number of processes that are poised to execute ⟦lines 156–160 of a `TSC`$(i,\cdot)$⟧⟨lines 219–221 of a `TWrite`$(i,\cdot)$⟩ operation. Process q's local queue $rsrvQ_q[i]$, for $i \in \{0,\ldots,m-1\}$, has initially $2n+4$ elements, and during each `updateQ`$(i,\cdot)$ one element is enqueued and one element is dequeued from $rsrvQ_q[i]$. Hence, the size of $rsrvQ_q[i]$ is $2n+4$ if q is not in the process of updating its queue, and $2n+5$ otherwise. Thus, at most $m(2n+5)$ tags can be in process q's queues, and so in total $nm(2n+5)$ of p's tags can be in all queues of all processes. Also s tags can be reserved but not in any of process' queues, because s processes can be poised to execute ⟦lines 156–160 of a `TSC`$(i,\cdot)$⟧⟨lines 219–221 of a `TWrite`$(i,\cdot)$⟩ operation at t_x. Hence in total, $nm(2n+5)+s$ of p's tags can be reserved at t_x. Since $s+\ell \le n$, the claim in (7.44) follows.

Next, we prove that

$$\begin{aligned}&\exists w \in \{p\beta,\ldots,(p+1)\beta-1\} \wedge w \neq x : \text{s.t. } b_w \text{ satisfies}\\&\text{the free condition throughout } [t_x,t_y).\end{aligned} \tag{7.45}$$

And so Part (a) follows.

As p owns $\beta = mn(2n+5)+\tau+3n+1$ tags, by (7.44) at least $2n+1$ of p's blocks have no tags active, announced, or reserved at t_x. By Invariant 7.21 and Invariant 7.22 (as it holds at

$t_x < t_f$), this implies that there exists a set B^* of $2n+1$ blocks that p owns, such that for each $b_z \in B^*$

$$\sigma(p,z) + \sum_{q=0}^{n-1} Act_q[z] = 0 \text{ and } \forall q \in \{0,\ldots,n-1\} : Emp_q[z] = 0 \text{ at } t_x. \tag{7.46}$$

Let b_z be some block in B^*. We show that

$$\sigma(p,z) + \sum_{q=0}^{n-1} Act_q[z] = 0 \text{ throughout } [t_x,t_y). \tag{7.47}$$

Proof of (7.47): We need to show that the value of $\sigma(p,z) + \sum_{q=0}^{n-1} Act_q[z]$ does not change during $[t_x,t_y)$. Suppose for the sake of contradiction that at some point during $[t_x,t_y)$, some process changes the value of $\sigma(p,z) + \sum_{q=0}^{n-1} Act_q[z]$, and let t be the first such point. This value can change in lines 199 and 202. Note that the value of $\sigma(p,z) + \sum_{q=0}^{n-1} Act_q[z]$ does not change in line 194, because we assume that at the same point when the value of $Act_q[z]$ is reset to 0 by p, for some q, the current value of $Act_q[z]$ gets added to $\sigma(p,z)$, and so the sum remains the same.

First suppose that at t, some process executes line 202 of a `Release`(g^*) operation Re, such that $g^* \in b_z$. Since we consider good transcripts, then g^* must be active right before $lin(Re) = t_{Re@202} = t$. By (7.46) and Invariant 7.22 at t_x, no tags of b_z is active at t_x. Thus, the owner of block b_z, process p, must have made $g^* \in b_z$ active at some point during $[t_x,t)$. Tag g^* becomes active when p's `GetFree()` operation returns g^*, so p's `GetFree()` returns g^* at some point during $[t_x,t)$. This is a contradiction, as we assumed p only returns tags of block b_x during $[t_x,t_y)$ and hence during $[t_x,t)$.

Next suppose that process p executes line 199, for some $tag_p = g^* \in b_z$. This implies that at t, process p returns g^*. However, this contradicts the assumption that p only returns tags of block b_x throughout $[t_x,t_y)$, and therefore at t. This completes the proof of (7.47).

Next we prove that there exist one block b_w in B^*, such that

$$Emp_q[w] = 0 \text{ for all } q \in \{0,\ldots,n-1\} \text{ throughout } [t_x,t_y). \tag{7.48}$$

Equations (7.47) and (7.48) prove (7.45) and so the claim follows.

Proof of (7.48): Here we show that each process q modifies Emp_q of at most two blocks in B^* throughout $[t_x, t_y)$, and as $|B^*| = 2n+1$, by (7.46) there is one block $b_w \in B^*$ such that $Emp_q[w] = 0$ for all $q \in \{0, \ldots, n-1\}$ throughout this interval. Suppose for the sake of contradiction that some process q modifies Emp_q of at least three blocks b_i, b_j, and b_k of B^* ($b_i \neq b_j \neq b_k$) during $[t_x, t_y)$. Let t_i, t_j, and t_k be the first points during $[t_x, t_y)$ at which process q modifies $Emp_q[i]$, $Emp_q[j]$, and $Emp_q[k]$, respectively, and assume w.l.o.g. that $t_i < t_j < t_k$. Therefore, by (7.46), $Emp_q[\ell] = 0$ throughout $[t_x, t_\ell)$ for $\ell \in \{i, j, k\}$. By Invariants 7.8–7.21, we know that

$$\text{no tag of block } b_k \text{ is announced or reserved during } [t_x, t_k). \tag{7.49}$$

Moreover, by Invariants 7.8–7.21, at any point $Emp_q[i] \geq 0$, $Emp_q[i] \geq 0$, and $Emp_q[k] \geq 0$. Thus,

$$q \text{ increases the value of } Emp_q[\ell] \text{ at } t_\ell, \text{ for any } \ell \in \{i, j, k\}. \tag{7.50}$$

Process q can only increment $Emp_q[\ell]$ in lines [[156, 165, 174, 182]]⟨219, 230, 241, 247⟩, or 206. In the following, we first show that

$$\begin{aligned} &q \text{ cannot execute any of lines [[156, 165, 174, 182]]⟨219, 230, 241, 247⟩ at } t_\ell, \\ &\text{where } \ell \in \{j, k\}. \end{aligned} \tag{7.51}$$

Next, we show that

$$q \text{ cannot execute line 206 at both } t_j \text{ and } t_k. \tag{7.52}$$

Hence, (7.51) and (7.52) together contradict (7.50), and thus that completes the proof of (7.48). This in addition to (7.47) proves (7.45), and so the claim follows.

Before we prove (7.51) and (7.52), note that by (7.46) and Invariant 7.22 at t_x, no tags of b_ℓ is active at t_x, for $\ell \in \{j, k\}$. Process p's `GetFree()` operations only return tags of $b_x \notin B^*$ (and so $b_x \neq b_\ell$) throughout $[t_x, t_y)$. Thus no tag of block b_ℓ is active during $[t_x, t_y)$ and so during $[t_x, t_\ell]$, because $t_\ell \in [t_x, t_y)$. Moreover, by (7.49) no tag of block b_ℓ is announced or reserved

during $[t_x, t_\ell)$. Hence by Corollary 7.10 and Claim 7.20, no tag of block b_ℓ is protected or stored in an element of A throughout $[t_x, t_\ell)$. Thus,

$$\text{no tag of block } b_\ell \text{ is active, protected, or stored in an element of } A \text{ throughout } [t_x, t_\ell), \text{ where } \ell \in \{j,k\}. \tag{7.53}$$

Now to prove (7.51), suppose for the sake of contradiction that q executes line $[\![156, 165, 174, \text{ or } 182]\!]\langle 219, 230, 241, \text{ or } 247\rangle$ at t_ℓ for some g^*, where $\ell \in \{j,k\}$, and $g^* \in b_\ell$. We show how each of these cases can yield a contradiction.

Case 1. $t_\ell = [\![t_{S@156}]\!]\langle t_{S@219}\rangle$, where S is a $[\![\texttt{TSC}(\cdot,(\cdot,g^*))]\!]\langle \texttt{TWrite}(\cdot,(\cdot,g^*))\rangle$ by q. Recall that in a good transcript tag g^* is occupied (i.e. g^* is either active, protected, or is stored in an element of A) throughout $[t_{S@inv}, lin(S))$. However, this contradicts (7.53), as $t_{S@inv} \leq t_\ell < lin(S)$.

Case 2. $t_\ell = [\![t_{S@165}]\!]\langle t_{S@230}\rangle$, where S is a $[\![\texttt{TSC}(\cdot,\cdot)]\!]\langle \texttt{TWrite}(\cdot,\cdot)\rangle$ by q, for $g' = g^*$. In this case, process q must have read $[\![(x^*, g^*, q) \text{ from } H[i][hCtr_q[i]] \text{ at } t_{S@157}]\!]\langle$the same value (x^*, g^*, c^*, b^*) from both $H[i][q][hCtr_q[i]]$ and $H[i][q][hCtr_q[i]]$ at $t_{S@220}$ and $t_{S@220}$, respectively$\rangle$. This implies that q has previously written this value and has not yet changed it. Therefore, g^* is reserved at $[\![t_{S@157}]\!]\langle t_{S@220}\rangle$ (by Claim 7.11). Since (7.49) says that no tag of b_ℓ is reserved during $[t_x, t_\ell)$, we have $[\![t_{S@157}]\!]\langle t_{S@220}\rangle < t_x$. Therefore, $[\![t_{S@157}]\!]\langle t_{S@220}\rangle < t_x < t_i < t_\ell = [\![t_{S@165}]\!]\langle t_{S@230}\rangle$. By (7.50) this implies that during $[\![(t_{S@157}, t_{S@165})]\!]\langle (t_{S@220}, t_{S@230})\rangle$, process q increments $Emp_q[i]$. However this is a contradiction as q does not increment $Emp_q[b]$ for any b during this interval.

Case 3. $t_\ell = [\![t_{L@174}]\!]\langle t_{L@241}\rangle$, where L is a $[\![\texttt{TLL}(\cdot)]\!]\langle \texttt{TRead}(\cdot)\rangle$ by q, for $g = g^*$. Then, process q must have read $[\![(\cdot, g^*)]\!]\langle (\cdot, g^*, \cdot)\rangle$ from $A[i]$ at $[\![t_{L@173}]\!]\langle t_{L@239}\rangle$. By Corollary 7.10, g^* is reserved at $[\![t_{L@173}]\!]\langle t_{L@239}\rangle$. This in addition to (7.49) implies that $[\![t_{L@173}]\!]\langle t_{L@239}\rangle < t_x$, and so $[\![t_{L@173}]\!]\langle t_{L@239}\rangle < t_x < t_i < t_\ell = [\![t_{L@174}]\!]\langle t_{L@241}\rangle$. Therefore during $[\![(t_{L@173}, t_{L@174})]\!]\langle (t_{L@239}, t_{L@241})\rangle$, process q increments $Emp_q[i]$ (see (7.50)), which is a

contradiction.

Case 4. $t_\ell = [\![t_{L@182}]\!]\langle t_{L@247}\rangle$, where L is a $[\![\texttt{TLL}(\cdot)]\!]\langle\texttt{TRead}(\cdot)\rangle$ by q, for $g' = g^*$. Process q must have read $[\![(\cdot, g^*, \cdot)$ from $H[i][q]$ in line 175$]\!]\langle$the same value (x^*, g^*, c^*, b^*) from both $H[i][q'][q]$ and $H[i][q'][q]$, for some q', in lines 242–243$\rangle$ of L, and so by Claim 7.11, tag g^* is reserved at $[\![t_{L@175}]\!]\langle t_{L@242}\rangle$. Since no tag of b_ℓ is reserved during $[t_x, t_\ell)$ by (7.49), we have $[\![t_{L@175}]\!]\langle t_{L@242}\rangle < t_x < t_i < t_\ell = [\![t_{L@182}]\!]\langle t_{L@247}\rangle$. Therefore, (7.50) requires q to increment $Emp_q[i]$ during $[\![(t_{L@175}, t_{L@182})]\!]\langle(t_{L@242}, t_{L@247})\rangle$, which is a contradiction.

This completes the proof of (7.51).

Now, we prove (7.52). Suppose for the sake of contradiction that q executes line 206 for some g_j and g_k at t_j and t_k, respectively, where $g_j \in b_j$ and $g_k \in b_k$. Recall that in a good transcript, when q executes `Protect(`g_k`)`, tag g_k must have been occupied at some point since q's last `Protect()` call. Process q's last `Protect()` call before t_k cannot be before t_j, as we assumed that q executes `Protect(`g_j`)` at t_j. However, by (7.53), no tag of block b_k is occupied throughout $[t_x, t_k)$ and so throughout $[t_j, t_k)$. This is a contradiction.

Proof of Part (b). As G returns the first tag of block b_y, we have $tag_p = y\delta$ at the response of G. Process p increments the value of tag_p in line 184 of each `GetFree()` operation. It can also change it to $(p\beta + j_p)\delta$ in line 196 of a `GetFree()` operation in case the if-condition in line 195 is satisfied. Suppose Part (b) does not hold, then the last time p modified tag_p before t_y is in line 184 of G. Hence, $tag_p = y\delta - 1$ just before it executes line 184 of G. Note that p writes a multiple of δ to tag_p in line 196. Thus, p does not execute this line during the last $\delta - 1$ `GetFree()` operations $G_1, \ldots, G_{\delta-1}$ that it executes before G as well as during G. This implies that each time p evaluates the if-condition in line 195 of each of $\{G_1, \ldots, G_{\delta-1}\} \cup G$, we have $\rho_p \neq 0$. I.e.

$$\rho_p \neq 0 \text{ when } p \text{ executes line 195 of any operation in } \{G_1, \ldots, G_{\delta-1}\} \cup G. \tag{7.54}$$

Note that by the implementation, p returns tags of block b_x starting from t_x and until before

G returns, and so tag_p belongs to block b_x until p increments it during G after which tag_p belongs to b_y. As each block contains δ tags, all $\delta - 1$ `GetFree()` operations by p before G (i.e. $\{G_1, \ldots, G_{\delta-1}\}$) return tags of block b_x and therefore

$$G_1 \text{ gets invoked after } t_x. \tag{7.55}$$

Process p increments (modulo $3n$) the value of its local ρ_p in line 189 of each `GetFree()` operation, and resets it to 0 in line 191 of a `GetFree()` when the if-condition in line 190 evaluates to true. Now we show that

$$p \text{ executes lines 191–192 during every } 2n \text{ consecutive operations in } \{G_1, \ldots, G_{\delta-n}\}. \tag{7.56}$$

Suppose not, then since p increments ρ_p (modulo $3n$) during each `GetFree()` operation, there is a `GetFree()` operation $G' \in \{G_1, \ldots, G_{\delta-n}\}$, such that $\rho_p = 2n$ at the response of G'. Then, after executing $n-1$ additional `GetFree()` operation, p executes a `GetFree()` operation $G'' \in \{G_1, \ldots, G_{\delta-1}\} \cup G$, such that $\rho_p = 3n - 1$ at the invocation of G'', and so $\rho_p = 0$ when p executes line 195 of G'' (because p increments ρ_p in line 189 of G''), which contradicts (7.54) and so (7.56) follows.

Now we show that

$$t_f > t_{G_{\delta-n}@rsp}. \tag{7.57}$$

By (7.56) when p evaluates the if-conditions of lines 190–197 while $\rho_p = 2n$ during $\{G_1, \ldots, G_{\delta-n}\}$, p resets this value to 0. Thus, p does not execute line 194 for $\rho_p = 2n$ during these operations, and so no cleaning interval starts during $\{G_1, \ldots, G_{\delta-n}\}$. The statement of (7.56) also implies that $\rho_p \in \{1, \ldots, 2n\}$ when p evaluates the if-conditions of lines 190–197 during $\{G_1, \ldots, G_{\delta-n}\}$. Therefore, no cleaning interval ends during these operations. Moreover, no cleaning interval starts after t_x and before G_1 (note that by (7.55) we have $t_x < t_{G_1@inv}$). This is because as we discussed it cannot end after the invocation of G_1, and if it ends during a `GetFree()` operation before the invocation of G_1, then p returns the first tag of a different

block than b_x in that operation, which is not the case, as by the assumption p only returns tags of b_x throughout $[t_x, t_y)$. So (7.57) follows.

Now we show how the assumption that p does not execute line 196 of G yields a contradiction. The value of j_p can only change in line 192 and line 197 of a `GetFree()` operation. Recall that by (7.54), p does not execute line 197 during $\{G_1, \ldots, G_{\delta-1}\} \cup G$. Thus, by (7.56) during operations in $\{G_1, \ldots, G_{\delta-n}\}$ the value of G changes only in line 192 of every $2n$ consecutive operations in this set of operations. Note that when p executes lines 191–192 its search interval just ends, and then in its next `GetFree()` it starts a new search interval, and the search interval continues for another $2n$ `GetFree()` operations. Therefore, after completing all operations in $\{G_1, \ldots, G_{\delta-n}\}$, p completes $\lfloor (\delta - n)/2n \rfloor = \beta$ search intervals and j_p is incremented modulo β before the beginning of each of these search intervals (line 192). I.e. p completes a search interval for each block that it owns (recall that p owns β blocks). Moreover, by (7.56), p finds either $sum_p \neq 0$ or $sum'_p \neq$ `false` at the end of each of these search intervals. As Invariant 7.22 holds throughout $[0, t_f]$ and by (7.57) t_f is after the response of $G_{\delta-n}$, we can apply Claim 7.27. Hence, none of β blocks that p owns is free throughout the interval that starts when G_1 gets invoked and ends by the response of $G_{\delta-n}$. However, by Part (a), process p owns a block b_w (possibly $w = y$), such that b_w is free throughout $[t_x, t_y)$, and by (7.55) and (7.57) throughout the interval that starts with the invocation of G_1 and ends with the response of $G_{\delta-n}$. This is a contradiction.

Proof of Parts (c), (d), and (e). To prove Part (c), we first show that there is a point $t_f < t_y$ at which p starts a cleaning interval of b_y, and a point $t_s < t_f$ at which p starts the last search interval of b_y before t_f.

Let j_p^* be the value of j_p when p executes line 196 during G, i.e. $y = p\beta + j_p^*$. By Part (b), $\rho_p = 0$ when p executes line 196 during G. As ρ_p gets incremented modulo $3n$ in line 189 of each `GetFree()`, we have $\rho_p = 3n - 1$ at the invocation of G. Process p can only reset the value of ρ_p in line 191 of a `GetFree()` operation when the if-condition in line 190 evaluates to true.

Since $\rho_p = 3n-1$ at the invocation of G, p executes $3n-1$ consecutive `GetFree()` operations $G_0,\ldots,G_{3n-2}$ right before G during which p does not reset ρ_p (in line 191), and $\rho_p = k$ at the invocation of G_k, for $k \in \{0,\ldots,3n-2\}$. I.e. for any $k \in \{0,\ldots,3n-2\}$, we know that

$$p\text{'s if-condition in line 190 of } G_k \text{ evaluates to false,} \tag{7.58}$$

and

$$\rho_p = k \text{ at the invocation of } G_k. \tag{7.59}$$

By (7.58), p does not change the value of j_p in line 192 of any of $G_0,\ldots,G_{3n-2}$. Moreover, by (7.59), $\rho_p \neq 0$ in line 195 of any of $G_0,\ldots,G_{3n-2}$, and so p does not change the value of j_p in line 197 of any of $G_0,\ldots,G_{3n-2}$. I.e.

$$j_p = j_p^* \text{ throughout } [t_{G_0@inv}, t_{G@197}). \tag{7.60}$$

Consider operation G_0, and recall that by (7.59), $\rho_p = 0$ at the invocation of this operation. Thus, p starts a search interval of $b_y = b_{p\beta+j_p^*}$ in line 185 of G_0 (because by (7.60), we have $j_p = j_p^*$ during this operation). Thus, we let t_s be $t_{G_0@185}$. Process p increments ρ_p during each $G_0,\ldots,G_{3n-2}$ (line 189). This search interval ends when p increments its ρ_p to $2n$ in line 189 of G_{2n-1}. By (7.58), in its next step p executes line 194 of G_{2n-1} for $j_p = j_p^*$, and so it starts a cleaning interval of $b_y = b_{p\beta+j_p^*}$. Hence, we let t_f be $t_{G_{2n-1}@194}$, and so $t_s < t_f < t_y$. This cleaning interval ends when p increments ρ_p to 0 in line 189 of G and therefore before t_y, which proves Part (d).

To finish the proof of Part (c), it only remains to show that $t_x < t_f$. In order to do so, we show $t_x < t_s = t_{G_0@185}$. Let G' be the `GetFree()` operation during which p returns the first tag of b_x at t_x. By Part (b), p executes line 196 during G'. Thus, $\rho_p = 0$ at the response of G', and $\rho_p = 3n-1$ at the invocation of this operation. Recall that $\rho_p = 3n-1$ at the invocation of G, and $\rho_p = k$ at the invocation of $G_0,\ldots,G_{3n-2}$ (by (7.59)), which are the last $3n-2$ `GetFree()` operations that p executes before G. Thus, p completes G' before $G_0,\ldots,G_{3n-2}$, and so $t_x < t_s = t_{G_0@185}$.

To prove Part (e), note that process p's `GetFree()` operations only return tags of b_x throughout $[t_x, t_y)$, and so throughout $[t_s, t_y)$ (because $t_x < t_s$). Thus, by Claim 7.28 all tags of b_y are free at throughout $[t_f, t_y)$, and so Part (e) follows. □

The following result comes immediately from Claim 7.29 and will be used later to prove Invariant 7.22.

Corollary 7.30. *Let b_y be a block owned by some process p, and t_f the point at which p starts a cleaning interval of b_y. If Invariant 7.22 holds throughout $[0, t_f]$, then all tags of b_y are free during this cleaning interval.*

Proof. By definition, any cleaning interval of b_y ends in line 189 of a `GetFree()` operation by p during which p returns the first tag of b_y. Thus by Claim 7.29 (e), all tags of b_x are free starting from t_f until just before the first tag of b_x is returned, and so throughout the cleaning interval. □

Using the results of Claim 7.23 to Claim 7.29, we can now prove the following.

Claim 7.31. *Consider a `GetFree()` operation G by some process p that returns g^* and suppose Invariant 7.22 holds throughout $[0, t_{G@199})$. Then tag g is free just before $lin(G)$.*

Proof. Let b_y be the block to which tag g^* belongs. By definition $lin(G) = t_{G@199} = t_{G@rsp}$. If p returns g^* from G, its local variable $tag_p = g^*$ at $lin(G)$. Process p modifies its local tag_p in two ways: it increments tag_p in line 184 of each `GetFree()` operation, and it may set tag_p to the first tag of a block in line 196 of a `GetFree()` operation. Thus, when p returns tag g^* at $lin(G)$, it must have returned the first tag of block b_y at some point $t_y \leq lin(G)$, and p does not return g^* from a `GetFree()` operation during $[t_y, lin(G))$. By Claim 7.29 all tags of b_y are free right before t_y, and so if $t_y = lin(G)$, then the claim follows. If $t_y < lin(G)$, then by Claim 7.24, tag g^* is free throughout $[t_y, lin(G))$ and so right before $lin(G)$. □

7.5.8 Proof of Invariant 7.22

Before we prove Invariant 7.22, we need to show one more property of the algorithm and that is the following. The value of $Act_p[x]$ does not change between the read and the write of this variable in lines 201–202 and lines 198–199.

Claim 7.32. *Consider some process p is executing lines 198–199 of a* `GetFree()` *operation G, or lines 201–202 of a* `Release()` *operation Re for $Act_p[x]$. Let Invariant 7.22 hold throughout $[0, t_{G@199})$ respectively $[0, t_{G@202})$. Then, the value of $Act_p[x]$ does not change during $[t_{G@198}, t_{G@199})$ and $[t_{G@201}, t_{G@202})$ respectively.*

Proof. The value of $Act_p[x]$ can only be modified by two processes, process p and the owner q (possibly $q = p$) of block b_x, where $q = \lfloor x/\beta \rfloor$; If $p \neq q$, then process p can change the value of $Act_p[x]$ only in line 202 of a `Release()` operation, and q can change the value of the same variable ($Act_p[x]$) in line 194 of a `GetFree()`. Otherwise, if $p = q$ (i.e. p is the owner of block b_x), then only p can change the value stored in $Act_p[x]$ and this can be done in line 202 of a `Release()`, and in line 194 and line 199 of a `GetFree()` operation.

First suppose that $p = q$, i.e. p is the owner of b_x. Thus, no other process can change the value of $Act_p[x]$ while p is executing lines 198–199 of a `GetFree()` operation, or lines 201–202 of a `Release()` operation, and so the claim follows.

Next suppose that $p \neq q$, i.e. p is not the owner of b_x. Suppose for the sake of contradiction that the claim does not hold. Since p is not the owner of block b_x, it cannot execute lines 198–199 of a `GetFree()` operation for $Act_p[x]$. Thus, p executes lines 201–202 of a `Release(`g^*`)` operation Re, such that $g^* \in b_x$, but the value of $Act_p[x]$ changes during $[t_{G@201}, t_{G@202})$. This must be because q (the owner of b_x) is executing line 194 of a `GetFree()` operation during this interval. Hence, $[t_{G@201}, t_{G@202})$ overlaps with a cleaning interval I of b_x by q. As I starts before $t_{G@202}$, and so Invariant 7.22 holds starting from point 0 and until at leastthe beginning of this cleaning interval, by Corollary 7.30, all tags of b_x are free throughout I. However, this is

a contradiction because we are considering only good transcripts, and so tag g^* must be active throughout $[t_{Re@inv}, lin(Re)] = [t_{Re@201}, t_{Re@202}]$. □

Now we can finally prove Invariant 7.22 (as repeated in the following).

Invariant 7.22 (Restated). *For every block b_x owned by some process p the following holds.*

$$\sigma(p,x) + \sum_{q=0}^{n-1} Act_q[x] = NumOfActive(x), \text{ and} \tag{7.61}$$

$$\sigma(p,x) = 0 \text{ at any point } t \text{ that is not during a cleaning interval of } b_x. \tag{7.62}$$

Proof. As no tag is active at the beginning, and $\sigma(p,x)$ and $Act_q[x]$ are initialized to 0, for all processes q, the invariant holds at time 0. Now assume for the sake of contradiction that the invariant holds throughout $[0,t)$, for some t, and at t some process takes one step such that the invariant does not hold right after t.

First suppose that (7.61) does not hold right after t. This implies that at t some process changes the value of $\sigma(p,x)$, $Act_q[x]$, or $NumOfActive(x)$, for some q. The value of $\sigma(p,x)$, $Act_q[x]$, or $NumOfActive(x)$ can change only in line 202, line 194 or line 199.

Consider the case in which process q executes line 202 of a `Release(`g^*`)` operation Re at $t = t_{Re@202}$, for $g^* \in b_x$. As the invariant holds just before t, by Claim 7.32, the value of $Act_q[x]$ gets decremented in line 202 at t. In a good transcript, tag g^* must be active right before $lin(Re) = t_{Re@202} = t$. Moreover g^* becomes inactive right after $lin(Re) = t$. Hence, the value of $NumOfActive(x)$ gets decremented by one at t. As the value of $\sigma(p,x)$ does not change at t, this implies that (7.61) holds right after t, which is a contradiction.

Next suppose that at t, process p executes line 194 for $\rho_p \bmod n = q$ and $p\beta + j_p = x$. This implies that the current value of $Act_q[x]$ gets added to $\sigma(p,x)$, and p writes 0 to $Act_q[x]$. Hence, $\sigma(p,x) + \sum_{q=0}^{n-1} Act_q[x]$ remains unchanged right after this point, and so (7.61) remains true right after t, which is a contradiction.

Finally consider the case in which the value of $Act_p[x]$ changes in line 199 of a `GetFree()` operation G by p, i.e. $t_= t_{G@199}$. Let g^* be the return value of G, then $g^* \in b_x$. The invariant holds just before t, hence by Claim 7.32, the value of $Act_p[x]$ gets incremented in line 199 at t. Moreover at point $lin(G) = t_{G@199} = t$, tag g^* is returned from G and therefore, G becomes active. By Claim 7.31 tag g^* is free, and so not active right before t. Thus the value of $NumOfActive(x)$ gets incremented by one at t. The value of $\sigma(p,x)$ does not change at t, therefore (7.61) holds right after t, which is a contradiction. This completes the proof of (7.61)

Next suppose that (7.62) does not hold right after t. Note that $\sigma(p,x)$ only changes during a cleaning interval of b_x. Thus, t must be the end of a cleaning interval of b_x, and let t_f be the point at which this cleaning interval starts. I.e. $\sigma(p,x) \neq 0$ right after the end of this cleaning interval at t.

By Corollary 7.30, all tags of b_x are free throughout $[t_f,t]$, which implies that none of them is active (i.e. $NumOfActive(x) = 0$) throughout this interval. Right before the cleaning interval of b_x starts at t_f, the invariant holds and so we have $\sigma(p,x) = 0$. Moreover, $NumOfActive(x) = 0$. Thus by (7.61), we have

$$\sum_{q=0}^{n-1} Act_q[x] = 0 \text{ right before } t_f. \tag{7.63}$$

As all tags of b_x are free throughout $[t_f,t]$, no process executes line 202 or line 199 during this interval, and so the value of $Act_q[x]$, for $q \in \{0,\ldots,n-1\}$, can only change in line 194 by p. Each time p writes 0 to $Act_q[x]$, for $q \in \{0,\ldots,n-1\}$, the current value of $Act_q[x]$ gets added to $\sigma(p,x)$. Thus, the value of $\sigma(p,x)$ at the end of this cleaning interval (i.e. at t) is the same as the value of $\sum_{q=0}^{n-1} Act_q[x]$ right before the beginning of this search interval (i.e. right before t_f), which by (7.63) is 0. This is a contradiction. □

7.5.9 Proof of Linearizability

Now we can prove Lemma 7.4 (from p. 192), as restated in the following for convenience.

Lemma 7.4 (Restated)**.** *If Λ is a good transcript, then*

(a) *the value of $A[i]$ changes to (x,g) at some point t, if and only if $t = lin(S)$, where S is a [[successful* $\texttt{TSC}(i,(x,g))$*]]⟨*$\texttt{TWrite}(i,(x,g))$*⟩ in Λ,*

(b) *if a [[*$\texttt{TLL}(i)$*]]⟨*$\texttt{TRead}(i)$*⟩ operation L in Λ returns (x,g), then $A[i] = (x,g)$ at $lin(L)$,*

(c) *if a* `GetFree()` *operation G in Λ returns g, then g is not occupied immediately before $lin(G)$ in Λ, and*

(d) *[[a* $\texttt{TSC}(i,\cdot)$ *operation S succeeds in Λ, if and only if there is no successful* $\texttt{TSC}(i,\cdot)$ *operation S', such that $lin(S') \in (lin(L), lin(S))$, where L is the last* $\texttt{TLL}(i)$ *by the same process before S in Λ.]]*

Proof. We prove each part separately.

Proof of Part (a). The value stored in $A[i]$ changes to (x,g) only in [[line 158 of a successful $\texttt{TSC}(i,(x,g))$]]⟨line 227 or line 232 of a $\texttt{TWrite}(i,(x,g))$⟩ operation, which by definition is the linearization point of that operation. Moreover, [[a $\texttt{TSC}(i,(x,g))$ operation S by some process p returns `true` only in line 168. This implies that p's $A[i].\texttt{SC}(x,g)$ in line 158 of S succeeds, and so the value of $A[i]$ changes to (x,g) at the linearization point of this operation.]]⟨a $\texttt{TWrite}(i,(x,g))$ operation always linearizes either in line 227 or line 232 depending on whether the if-condition in line 224 evaluates to true or false, and that is the point at which the value of $A[i]$ changes to (x,g).⟩

Proof of Part (b). Let q be the process that executes L. First suppose L is a direct [[$\texttt{TLL}()$]]⟨$\texttt{TRead}()$⟩. As L returns (x^*,g^*), q must have read [[(x^*,g^*)]]⟨$(x^*,g^*,\cdot)$⟩ from $A[i]$ in line 173 of L at $lin(L) =$[[$t_{L@173}$]]⟨$t_{L@239}$⟩. Now, suppose L is an indirect [[$\texttt{TLL}()$]]⟨$\texttt{TRead}()$⟩. By Claim 7.14, there exists a [[$\texttt{TSC}(i,(x^*,g^*))$]]⟨$\texttt{TWrite}(i,(x^*,g^*))$⟩ operation S by some process p during which p successfully writes [[(x^*,g^*)]]⟨(x^*,g^*,p)⟩ to $A[i]$ at $lin(S)$. In this case [[and if q sets $flag_q[i]$ in line 181 of L]], then $lin(L) = lin(S)$, and so $A[i]$ has value [[(x^*,g^*)]]⟨(x^*,g^*,p)⟩ at $lin(L)$. [[If L is indirect and q does not set the flag, then process q reads the return value of L, (x^*,g^*), from $A[i]$ at $t_{L@179} = lin(L)$.]]

Proof of Part (c). By Claim 7.31, tag g^* is free and so not occupied right before $lin(G) = t_{G@199} = t_{G@rsp}$.

Proof of Part (d). Let p be the process which executes S. Note that process p changes the value of its local variable $flag_p[i]$ only during a `TLL(i)` operation, and more specifically, it sets it to `false` at the beginning of each `TLL(i)` and sets it to `true` only if its if-condition in line 180 evaluates to `true`. Thus,

$$flag_p[i] = 1 \text{ at the invocation of } S \text{ if and only if } p \text{ sets } flag_p[i] \text{ in line 181 of } L. \tag{7.64}$$

First suppose S succeeds (i.e. it returns `true` in line 168). Then, we have $flag_p[i] \neq 1$ at the invocation of S, and by (7.64) p does not set $flag_p[i]$ in line 181 of L. Thus, $lin(L) = t_{L@173}$ if L is a direct `TLL()`, and $lin(L) = t_{L@179}$ if L is an indirect `TLL()`. Moreover, as S returns `true`, by definition, $lin(S)$ is when p successfully executes $A[i]$.`SC()` in line 158. Thus no process executes a successful $A[i]$.`SC()` operation between p's last $A[i]$.`LL()` and $lin(S)$. Process p only executes $A[i]$.`LL()` in lines 173 and 179 of a `TLL(i)`, hence the last $A[i]$.`LL()` that p executes before S is at $t_{L@173} = lin(L)$ if L is a direct `TLL()`, or is at $t_{L@179} = lin(L)$ if L is an indirect `TLL()`. This completes the proof of this case.

Now suppose S does not succeed, i.e. it returns `false` either in line 155 or line 160. First assume S returns in line 155, so $flag_p[i] = 1$ at the invocation of S. By (7.64), p sets $flag_p[i]$ to `true` in line 181 of L. I.e. $(x',g') \neq (x'',g'')$ when p executes 180 of L, where $(x',g',\cdot)$ is the value p reads from $H[i][p]$ in line 175, and (x'',g'') is the value p reads from $A[i]$ in line 179 of L. As p executes line 181 during L, $lin(L)$ is the point when some process writes (x',g') to $A[i]$ at some point before $t_{L@175}$ (by Claim 7.14 such a point exists). Moreover, $A[i] = (x'',g'') \neq (x',g')$ at $t_{L@179}$. Thus, by Part (a) some successful `TSC(i,·)` has changed the value of $A[i]$ after $lin(L)$ and before $t_{L@179}$. Hence, a successful `TSC(i,·)` linearizes at some point during $[lin(L), t_{L@179}] \subseteq [lin(L), lin(S)]$.

Next assume S returns in line 160. So $flag_p[i] \neq 1$ at the invocation of S, and so by (7.64), p does not set $flag_p[i]$ in line 181 of L. Thus, $lin(L) = t_{L@173}$ if L is a direct `TLL()`, and

$lin(L) = t_{L@179}$ if L is an indirect TLL(). Moreover, since S returns in line 160, p's $A[i]$.SC() in line 158 of S fails. So some process executes a successful $A[i]$.SC() operation between p's last $A[i]$.LL() and $lin(S) = t_{L@rsp}$. As p only executes $A[i]$.LL() in lines 173 and 179 of a TLL(i), the last $A[i]$.LL() that p executes before S is at $t_{L@173} = lin(L)$ if L is direct, or is at $t_{L@179} = lin(L)$ if L is indirect. Hence, some process executes a successful $A[i]$.SC() operation during $[lin(L), lin(S)]$. □

Lemma 7.33. *The implementation of [[TLSA]]⟨TRA⟩ as provided in [[Figure 7.9]]⟨Figure 7.11⟩ and Figure 7.10 is linearizable.*

Proof. Consider a transcript Λ obtained from the implementation of [[TLSA]]⟨TRA⟩ given in [[Figure 7.9]]⟨Figure 7.11⟩ and Figure 7.10. Let $H = \Gamma(\Lambda)$, and $\mathcal{L}(H)$ be the sequential history obtained from H, by ordering all operations M in H according to their $lin(M)$ values. If Λ is not a good transcript, then by Claim 7.3, H has a linearization, which proves the claim.

Now suppose Λ is a good transcript. By Corollary 7.5 the sequential history obtained by mapping each operation M in H to its $lin(M)$ is valid. So it just remains to show that $lin(M)$ is between the invocation and the response of M, i.e. $lin(M) \in [t_{M@inv}, t_{M@rsp}]$.

If M is any of [[TSC()]]⟨TWrite()⟩, GetFree(), Release(), Unprotect(), and CancelProtect(), then this follows immediately from the definition of $lin(M)$, which assigns this point to a line of the code of M. Now consider a [[TLL()]]⟨TRead()⟩ operation M by some process q which returns (x^*, g^*). If M is a direct operation (i.e. q' if-condition in [[line 176]]⟨line 244⟩ evaluates to false), then $lin(M) =$[[$t_{M@173}$]]⟨$t_{M@239}$⟩$\in [t_{M@inv}, t_{M@rsp}]$. If M is an indirect operation (i.e. q' if-condition in [[line 176]]⟨line 244⟩ evaluates to true), then by Claim 7.14, there exists a [[successful TSC()]]⟨TWrite()⟩ operation S by some process p during which p writes [[(x^*, g^*)]]⟨(x^*, g^*, p)⟩ to $A[i]$ at $lin(S)$. In this case [[and if q sets $flag_q[i]$ in line 181 of M]], $lin(M) = lin(S)$, and by Claim 7.14(b), $lin(S) \in [t_{M@inv}, t_{M@rsp}]$. [[If M is indirect and q does not set the flag, then $lin(M) = t_{M@179} \in [t_{M@inv}, t_{M@rsp}]$.]] □

7.5.10 Proof of Theorem 7.1

By Lemma 7.33, both proposed implementations of TRA and TLSA are linearizable. By Observation 7.12 the loop in the `TLL()` operation ends in a constant number of steps. Hence, it follows immediately from the pseudocode that all operations of both implementations have constant step complexity.

The number of blocks we use for each process is β, and so the total number of blocks is $n\beta$, where $\beta = mn(2n+5) + \tau + 3n + 1 = O(mn^2 + \tau)$. Each block contains δ tags, where $\delta = 2\beta n + n$. Hence, the total number of tags is $\Delta(m,n,\tau) = n\beta\delta = n\beta(2\beta n + n) = O(m^2n^6 + n^2\tau^2 + mn^4\tau)$.

The implementation of TLSA uses $O(mn)$ shared LL/SC objects. Moreover, the number of shared registers used in both TLSA and TRA is dominated by the number of ABA-detecting registers in array Emp, which is $O(n^2\beta)$. By Theorem 5.1, each ABA-detecting register can be implemented with $O(n)$ read-write registers. Thus, the total number of shared objects TLSA and TRA use is $M(m,n,\tau) = O(n^3\beta) = O(mn^5 + n^3\tau)$.

Each register of arrays Emp and Act needs to store $\log(mn + \tau)$ bits, and each ⟦LL/SC object in array A, as well as $H[i]$⟧⟨register in array A, as well as $H[i][j]$ and $H'[i][j]$⟩ need to store only $O(\log(\Delta(m,n,\tau)))$ bits in addition to the data they need to store. All other registers need at most $O(\log(\Delta(m,n,\tau)))$ many bits. This concludes the proof of Theorem 7.1.

Chapter 8

Summary and Future Work

In this work, we presented multiple transformations as well as implementations of new primitives in the standard shared memory model. These implementations empower algorithm designers with additional or stronger tools to tackle problems that do not have trivial solutions using bounded space, and could have a considerable impact on the design and complexity analysis of shared memory algorithms. The proofs for most of our implementations are very technical, but once obtained, the resulting objects can be easily employed in a variety of contexts in a black box fashion.

Selected Techniques. The algorithms of Chapters 4 and 7 benefit from reclaiming objects once it is ensured that no process will access those objects. However, asynchrony makes this task quite challenging, because there is no bound on how far behind a process may be compared to other processes in the system. In particular, it is not always known to other processes which object a process is about to access in its next step. Therefore, reclaiming any object may cause unsafe access of a memory space.

A common way to deal with this problem is using the general announcement technique: After a process p reads the address x of an object $O[x]$ from a register X, it announces value x by writing it into an announce array entry. The purpose of this technique is to let other processes know that p is about to access this object. So before any process tries to reclaim an object, it has to ensure no process currently has this object announced. Since it may take process p too long to announce x, in most existing memory reclamation solutions, such as Hazard Pointers (Michael, 2004b), to ensure a safe access to object $O[x]$, process p has to read register X again. The process only proceeds to access object $O[x]$ if X still holds value x. Otherwise, the process has to start over, because the memory at address x may have been reclaimed. This technique

for memory reclamation can lead to an unbounded number of restarts of a process's operation.

Instead of starting over, in Chapters 4 and 7, we devise a helping technique called hint mechanism, that allows a wait-free derefrencing: Some process helps process p by providing an alternative object, called a hint, that process p can access safely, in case it takes process p too long to make its announcement. Process p uses a hint over the value it reads from X only if it can ensure that the hint was provided since it started its current operation call. Our algorithm guarantees that either such a hint is available for p, or process p's announcement was timely, that is the location x it intends to access has not been reclaimed.

To achieve both linearizability and safety, the process that provides the hint has to ensure that the hint is indeed stored in X at some point during p's current operation call, and moreover, no process reclaims the hint object as long as process p could possibly access this object. Depending on what base objects are available, the hint technique is implemented differently. When using only registers as base objects, a sequence number is employed to synchronize between the provider and receiver of the hint. We bound the value of this sequence number in an adhoc manner by resetting the hints with some calculated frequency and deliberately maintaining a second copy of each hint. Using LL/SC as base objects, processes synchronize by allowing the receiver of the hint to reset a LL/SC object to indicate the start of its operation. Then a hint is only provided if the LL/SC object stores the initial value. In both cases, each process has to keep track of the hints it currently has provided and that can potentially still be used by other processes as non-reclaimable.

Another component of announcement technique requires that a process reads an entire array of size n to reclaim one object. Earlier research (Michael, 2004b) reduced the cost by reading the announce array once for every $O(n)$ reclaimed objects, to achieve a low amortized step complexity. In this work, we reduce the low amortized to worst case constant step complexity by proposing a deamortization technique. Each process is responsible for a set of objects, and it utilizes local data structures of polynomial size (in n) to keep track of those objects that the process finds

non-reclaimable at each point. The same technique is used for providing and maintaining hints, as well.

We also use the announcement technique to design an ABA-detecting register from bounded registers in Chapter 5. However, with a careful comparison of the value of the object before and after the announcement and during the previous `DRead()` operation, the process can decide the return value of its current `DRead()` call.

In the main algorithm of Chapter 6, we observe that any `TAS()` call that overlaps a `Reset()` operation call can immediately return 1 (and lose). We use an ABA-detecting 1-bit register to indicate whether a `Reset()` call is pending. Thus, processes keep reading this register after executing every shared memory step, and they decide based on the value they read to either continue competing or just lose their `TAS()` call. Moreover, since only the winner of a `TAS()` call can execute a `Reset()` operation, by allowing each `Reset()` call to linearize at its response, we can ensure that no two `Reset()` calls can overlap. These two ideas together allow us to obtain long-lived TAS implementations with improved step and space complexity than the ones we can achieve with the general transformation of Chapter 4.

In Chapter 7, in addition to the announcement and helping techniques, we use an information aggregation technique. The aggregated information helps the processes to decide about a big block of tags, rather than dealing with each tag individually. Although this significantly improves the step complexity of the implementation, it increases the number of tags used and so the space requirement of this implementation. ABA-detecting registers and a double collect approach (Afek, Attiya, Dolev, Gafni, Merritt and Shavit, 1993) are used to scan the information that all processes maintain.

Results and future directions. Our general transformation for augmenting objects with a concurrent `Write()` operation, works for any writable type, provided that there is an implementation that supports non-concurrent writes. This transformation adds only a constant number of steps overhead, and uses polynomially many instances of the original object. To the best of our

knowledge this is the first such general transformation from bounded space with constant step complexity.

We also introduced several efficient implementations of long-lived test-and-set objects from registers. As one of the results, we obtained the first long-lived test-and-set implementation that has optimal (linear) space complexity, and at the same time yields sub-logarithmic expected step complexity for both, `TAS()` and `Reset()` operations (against the oblivious adversary). Our techniques heavily rely on the property that only the winner of a `TAS()` call can reset the object. It would be interesting to research whether similar techniques can be applied to obtain other long-lived objects from one-time ones, or whether restricted forms of `Write()` operations can be implemented more space efficiently than the ones presented in Chapter 4. Another research direction might be to improve the combined space used by multiple long-lived TAS objects. It is conceivable that instead of a multiplicative overhead in the space, one can achieve k randomized wait-free long-lived TAS objects from $O(k+n)$ one-time ones.

All our transformations are wait-free, and in our examples they are applied to wait-free implementations, and the resulting objects are also wait-free. We believe that our transformations preserve all progress conditions. It would be interesting to prove that this is in fact the case. In general, it would be worthwhile to specify what kind of transformations preserve the progress condition of the original objects.

Moreover, this work suggests a new primitive, an ABA-detecting register, which provides a simple and efficient solution to the ABA problem, without incurring a step complexity overhead beyond a constant additive term. Using this object, algorithm designers can focus on finding solutions for their main problem without being concerned about ABAs.

This work provides the first study on the complexity of tagging, and showed that tags can be bounded with only constant time overhead. Using our taggable object abstraction for memory reclamation can, in some cases, lead to wait-free read-only operations, where other memory reclamation techniques yield only lock-freedom. We see it as an important contribution of this

work to present a useful abstraction, and provide building blocks that can be easily reused.

We show that our new primitives have efficient implementations with constant step complexity. Even though the space requirement of our implementations is polynomial (in several parameters, including the number of processes), it may be too high for practical use. The main goal of this research is not to provide a finished practical system, but to simplify future attempts to find solutions with bounded space for algorithmic problems, by using the primitives presented here, and do not have to find ad-hoc solutions for bounding tags. Also, we believe this is the beginning of a new line of research to find more space efficient implementations of this useful abstraction.

Bibliography

Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit (1993). Atomic snapshots of shared memory. *Journal of the ACM*, vol. 40, pp. 873–890.

Y. Afek, E. Gafni, J. Tromp and P. Vitányi (1992). Wait-free test-and-set (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, pp. 85–94.

Z. Aghazadeh, W. Golab and P. Woelfel (2013). Brief announcement: resettable objects and efficient memory reclamation for concurrent algorithms. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 322–324.

Z. Aghazadeh, W. Golab and P. Woelfel (2014). Making objects writable. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 385–395.

Z. Aghazadeh and P. Woelfel (2014). Space- and time-efficient long-lived test-and-set objects. In *Proceedings of 18th International Conference On Principles Of DIstributed Systems (OPODIS)*, pp. 404–419.

Z. Aghazadeh and P. Woelfel (2015). On the time and space complexity of ABA prevention and detection. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 193–202.

Z. Aghazadeh and P. Woelfel (2016). Upper bounds for boundless tagging with bounded objects. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*, pp. 442–457.

D. Alistarh and J. Aspnes (2011). Sub-logarithmic test-and-set against a weak adversary. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC)*, pp. 97–109.

D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert and M. Zadimoghaddam (2011a). Optimal-time adaptive strong renaming, with applications to counting. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 239–248.

D. Alistarh, J. Aspnes, S. Gilbert and R. Guerraoui (2011b). The complexity of renaming. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 718–727.

D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu and R. Guerraoui (2010). Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, pp. 94–108.

D. Alistarh, K. Censor-Hillel and N. Shavit (2016). Are lock-free concurrent algorithms practically wait-free? *Journal of the ACM*, vol. 63, pp. 31:1–31:20.

D. Alistarh, W. M. Leiserson, A. Matveev and N. Shavit (2017). Forkscan: Conservative memory reclamation for modern operating systems. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pp. 483–498.

J. H. Anderson and M. Moir (1995). Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 184–193.

M. Arbel-Raviv and T. Brown (2017). Reuse, don't recycle: Transforming lock-free algorithms that throw away descriptors. *ArXiv e-prints*, vol. abs/1708.01797.

J. Aspnes, M. Herlihy and N. Shavit (1994). Counting networks. *Journal of the ACM*, vol. 41, pp. 1020–1048.

H. Attiya and O. Rachman (1998). Atomic snapshots in o(n log n) operations. *SIAM Journal on Computing*, vol. 27, pp. 319–340.

H. Attiya and J. Welch (2004). *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons.

O. Balmau, R. Guerraoui, M. Herlihy and I. Zablotchi (2016). Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 349–359.

A. Braginsky, A. Kogan and E. Petrank (2013). Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the 25th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 33–42.

T. Brown, F. Ellen and E. Ruppert (2013). Pragmatic primitives for non-blocking data structures. In *Proceedings of the 32th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 13–22.

T. Brown, F. Ellen and E. Ruppert (2014). A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 329–342.

T. A. Brown (2015). Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 261–270.

H. Buhrman, A. Panconesi, R. Silvestri and P. Vitányi (2006). On the importance of having an identity or, is consensus really universal? *Distributed Computing*, vol. 18, pp. 167–176.

J. Burns and N. Lynch (1993). Bounds on shared memory for mutual exclusion. *Information and Computation*, vol. 107, pp. 171–184.

K. Censor-Hillel, E. Petrank and S. Timnat (2015). Help! In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 241–250.

T. Z. Chen and Y. Wei (2016). Step optimal implementations of large single-writer registers. In *Proceedings of 20th International Conference On Principles Of DIstributed Systems (OPODIS)*, pp. 32:1–32:16.

A. Clements, F. Kaashoek and N. Zeldovich (2012). Scalable address spaces using RCU balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 199–210.

N. Cohen and E. Petrank (2015). Automatic memory reclamation for lock-free data structures. In *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*, pp. 260–279.

D. Detlefs, P. Martin, M. Moir and G. Steele Jr. (2002). Lock-free reference counting. *Distributed Computing*, vol. 15, pp. 255–271.

S. Doherty, M. Herlihy, V. Luchangco and M. Moir (2004). Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 31–39.

D. Dolev and N. Shavit (1997). Bounded concurrent time-stamping. *SIAM Journal on Computing*, vol. 26, pp. 418–455.

C. Dwork, M. Herlihy and O. Waarts (1993). Bounded round numbers. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 53–64.

C. Dwork and O. Waarts (1992). Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 655–666.

C. Dwork and O. Waarts (1999). Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *Journal of the ACM*, vol. 46, pp. 633–666.

W. Eberly, L. Higham and J. Warpechowska-Gruca (1998). Long-lived, fast, waitfree renaming with optimal name space and high throughput. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pp. 149–160.

F. Ellen, Y. Lev, V. Luchangco and M. Moir (2007). Snzi: scalable nonzero indicators. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 13–22.

F. Ellen and P. Woelfel (2013). An optimal implementation of fetch-and-increment. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, pp. 284–298.

P. Fatourou and N. D. Kallimanis (2011). A highly-efficient wait-free universal construction. In *Proceedings of the 23rd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 325–334.

F. E. Fich, D. Hendler and N. Shavit (2006). On the inherent weakness of conditional primitives. *Distributed Computing*, vol. 18, pp. 267–277.

K. Fraser (2004). *Practical lock-freedom*. PhD thesis, University of Cambridge.

G. Giakkoupis, M. Helmi, L. Higham and P. Woelfel (2013). An $O(\sqrt{n})$ space bound for obstruction-free leader election. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, pp. 46–60.

G. Giakkoupis, M. Helmi, L. Higham and P. Woelfel (2015). Test-and-set in optimal space. In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 615–623.

G. Giakkoupis and P. Woelfel (2012). On the time and space complexity of randomized test-and-set. In *Proceedings of the 31th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 19–28.

G. Giakkoupis and P. Woelfel (2014). Randomized mutual exclusion with constant amortized RMR complexity on the DSM. In *Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 504–513.

A. Gidenstam, M. Papatriantafilou, H. Sundell and P. Tsigas (2009). Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 1173–1187.

W. Golab (2010). *Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations*. PhD thesis, University of Toronto.

W. Golab, V. Hadzilacos, D. Hendler and P. Woelfel (2012). RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, vol. 25, pp. 109–162.

W. Golab, L. Higham and P. Woelfel (2011a). Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC)*, pp. 373–382.

W. Golab, L. Higham and P. Woelfel (2011b). Linearizable implementations do not suffice for randomized distributed computation. *ArXiv e-prints*, vol. abs/1103.4690.

S. Haldar and P. M. B. Vitányi (2002). Bounded concurrent timestamp systems using vector clocks. *Journal of the ACM*, vol. 49, pp. 101–126.

T. E. Hart, P. E. McKenney, A. D. Brown and J. Walpole (2007). Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, vol. 67, pp. 1270–1285.

D. Hendler, N. Shavit and L. Yerushalmi (2010). A scalable lock-free stack algorithm. *Journal of Parallel and Distributed Computing*, vol. 70, pp. 1–12.

M. Herlihy (1988). Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 276–290.

M. Herlihy (1990). A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 197–206.

M. Herlihy (1991). Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 124–149.

M. Herlihy, V. Luchangco, P. A. Martin and M. Moir (2005). Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems*, vol. 23, pp. 146–196.

M. Herlihy, V. Luchangco and M. Moir (2002). The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, pp. 339–353.

M. Herlihy and N. Shavit (2008). *The art of multiprocessor programming*. Morgan Kaufmann.

M. Herlihy and J. Wing (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, vol. 12.

J.-H. Hoepman (1999). Long-lived test-and-set using bounded space. Technical Report, University of Twente.

IBM (1983). IBM system/370 extended architecture, principles of operation. Technical Report, IBM. Publication No. SA22-7085.

A. Israeli and M. Li (1987). Bounded time-stamps. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 371–382.

A. Israeli and M. Pinhasov (1992). A concurrent time-stamp scheme which is linear in time and space. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, pp. 95–109.

A. Israeli and L. Rappoport (1994). Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 151–160.

P. Jayanti (1998). A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pp. 216–230.

P. Jayanti and S. Petrovic (2003). Efficient and practical constructions of LL/SC variables. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 285–294.

P. Jayanti, K. Tan and S. Toueg (2000). Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, vol. 30, pp. 438–456.

A. Kogan and E. Petrank (2011). Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 223–234.

A. Kogan and E. Petrank (2012). A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 141–150.

C. Kruskal, L. Rudolph and M. Snir (1988). Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, vol. 10, pp. 579–601.

E. Ladan-Mozes and N. Shavit (2008). An optimistic approach to lock-free FIFO queues. *Distributed Computing*, vol. 20, pp. 323–341.

L. Lamport (1974). A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, vol. 17, pp. 453–455.

L. Lamport (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, vol. 28, pp. 690–691.

L. Lamport (1986). On interprocess communication. part I: basic formalism. *Distributed Computing*, vol. 1, pp. 77–85.

A. Larsson, A. Gidenstam, P. H. Ha, M. Papatriantafilou and P. Tsigas (2008). Multiword atomic read/write registers on multiprocessor systems. *ACM Journal of Experimental Algorithmics*, vol. 13.

H. Lee (2010). Fast local-spin abortable mutual exclusion with bounded space. In *Proceedings of 14th International Conference On Principles Of DIstributed Systems (OPODIS)*, pp. 364–379.

M. Loui and H. Abu-Amara (1987). Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, vol. 4, p. 31.

P. E. McKenney and J. Slingwine (1998). Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pp. 509–518.

M. Michael (2002). High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 73–82.

M. Michael (2004a). ABA prevention using single-word instructions. Technical Report, IBM T. J. Watson Research Center.

M. Michael (2004b). Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, pp. 491–504.

M. Michael (2004c). Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the 18th International Symposium on Distributed Computing (DISC)*, pp. 144–158.

M. Michael (2004d). Scalable lock-free dynamic memory allocation. In *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 35–46.

M. Michael and M. L. Scott (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 267–275.

M. Moir (1997). Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 219–228.

A. Panconesi, M. Papatriantafilou, P. Tsigas and P. Vitányi (1998). Randomized naming using wait-free shared variables. *Distributed Computing*, vol. 11, pp. 113–124.

G. Peterson (1983). Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 46–55.

S. A. Plotkin (1989). Sticky bits and universality of consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 159–175.

S. Prakash, Y. Lee and T. Johnson (1991). A non-blocking algorithm for shared queues using compare-and-swap. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pp. 68–75.

Y. Riany, N. Shavit and D. Touitou (2001). Towards a practical snapshot algorithm. *Theoretical Computer Science*, vol. 269, pp. 163–201.

C. Shann, T. Huang and C. Chen (2000). A practical nonblocking queue algorithm using compare-and-swap. In *Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 470–475.

N. Shavit and A. Zemach (1996). Diffracting trees. *ACM Transactions on Computer Systems*, vol. 14, pp. 385–428.

V. Shikaripura and A. D. Kshemkalyani (2002). A simple, memory-efficient bounded concurrent timestamping algorithm. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation (ISAAC)*, pp. 550–562.

J. M. Stone (1990). A simple and correct shared-queue algorithm using compare-and-swap. In *Proceedings of Supercomputing*, pp. 495–504.

E. Styer and G. Peterson (1989). Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 177–192.

H. Sundell (2005). Wait-free reference counting and memory management. In *Proceedings of 19th International Parallel and Distributed Processing Symposium (IPDPS)*.

S. Timnat, A. Braginsky, A. Kogan and E. Petrank (2012). Wait-free linked-lists. In *Proceedings of 16th International Conference On Principles Of DIstributed Systems (OPODIS)*, pp. 330–344.

S. Timnat and E. Petrank (2014). A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 357–368.

J. Tromp and P. Vitányi (2002). Randomized two-process wait-free test-and-set. *Distributed Computing*, vol. 15, pp. 127–135.

P. Tsigas and Y. Zhang (2001). A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 134–143.

J. Valois (1995). Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 214–222.

P. M. B. Vitányi and B. Awerbuch (1986). Atomic shared register access by asynchronous hardware (detailed abstract). In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 233–243.

P. Woelfel (2017). Algorithms for distributed computation. CPSC 561/661 Lecture Notes.

L. Zhu and F. Ellen (2015). Atomic snapshots from small registers. In *Proceedings of 19th International Conference On Principles Of DIstributed Systems (OPODIS)*, pp. 17:1–17:16.

Index

O

R

S

T

W