CONSTANT-RMR IMPLEMENTATIONS OF CAS AND OTHER SYNCHRONIZATION PRIMITIVES USING READ AND WRITE OPERATIONS

by

Wojciech Golab

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2010 by Wojciech Golab
Abstract

Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations

Wojciech Golab
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2010

We consider asynchronous multiprocessors where processes communicate only by reading or writing shared memory. We show how to implement consensus, all comparison primitives (such as CAS and TAS), and load-linked/store-conditional using only a constant number of remote memory references (RMRs), in both the cache-coherent and the distributed-shared-memory models of such multiprocessors. Our implementations are blocking, rather than wait-free: they ensure progress provided all processes that invoke the implemented primitive are live.

Our results imply that any algorithm using read and write operations, comparison primitives and load-linked/store-conditional, can be simulated by an algorithm that uses read and write operations only, with at most a constant-factor increase in RMR complexity.
Acknowledgements

I am deeply indebted to my supervisor, Prof. Vassos Hadzilacos, for his patient guidance during my PhD studies, his attention to detail, and his constructive criticism. Sincere thanks to Prof. Angela Demke-Brown and Prof. Sam Toueg for their long-standing service on my thesis committee and their helpful feedback on my work. Many thanks also to Prof. Allan Borodin and Prof. Prasad Jayanti for taking part in my SGS defence and for their insightful comments.

I would also like to acknowledge my collaborators in this research and thank them for many stimulating discussions and for engaging me in their work – Prof. Vassos Hadzilacos, Prof. Danny Hendler and Prof. Philipp Woelfel. In addition to supervising my thesis, Prof. Hadzilacos contributed to the introduction and the high-level descriptions of several algorithms, found optimizations related to the name consensus algorithm (see Chapter 3), helped formalize the locality properties for the cache-coherent model (see Chapter 7), and also suggested the use of the consensus algorithm from Chapter 3 in the universal construction presented in Chapter 10. Prof. Hendler originally motivated the idea of an RMR hierarchy for shared object implementations, introduced the locality property for the DSM model, and was active in discussions regarding how Compare-And-Swap can be implemented using repeated leader election. Prof. Woelfel proposed solving name consensus using leader election as a “black box” (see Chapter 3). He also proposed a way to make the implementation of ECAS writable.

Sincere thanks also to the anonymous referees for their detailed feedback on publications arising from this work, as well as to Prof. Faith Ellen, Robert Danek, George Giakkoupis and Stephanie Horn for their feedback on trial runs of conference presentations.

Finally, I would like to thank the administrative staff at the university of Toronto – Julie Weedmark, Linda Chow and Kolden Simmonds – and all my fellow graduate students for helping make my experience as a PhD student delightful.
Dedication

To my parents and brother for their endless support.
# Contents

1 Introduction  

2 Model of Computation and Definitions  

3 Consensus  
   3.1 Name Consensus in the DSM Model: A High-Level Description  
   3.2 Instrumented Leader Election Algorithm  
   3.3 Handshaking Protocol  
   3.4 Signalling Mechanism  
   3.5 Name Consensus in the DSM Model: A Detailed Description  
   3.6 Conclusion  

4 Pseudo-Locks  
   4.1 Pseudo-locks in the CC Model  
   4.2 Pseudo-lock Implementation for the DSM Model  
   4.3 Conclusion  

5 Block Manager  
   5.1 Linearizable $O(1)$-RMR Implementation  
   5.2 Conclusion  

6 Extended Compare-and-Swap  
   6.1 Linearizable $O(1)$-RMR Implementation of ECAS  
       6.1.1 Analysis  
   6.2 Conclusion  

7 Locally-Accessible Implementations  
   7.1 Locally-Accessible CAS and LL/SC  

v
12 Space Complexity in Locally-Accessible ECAS

12.1 Write-Back CC Model .............................................. 186
12.2 Write-Through CC Model ........................................ 189
  12.2.1 Implementations $I^B_E$ and $I^B_{EW}$ ....................... 191
  12.2.2 Implementations $I^{BL}_E$ and $I^{BL}_{EW}$ .................. 196
12.3 DSM Model ....................................................... 203
  12.3.1 Non-special Allocator ....................................... 205
  12.3.2 Special Allocator ........................................... 209
12.4 Conclusion ....................................................... 229

13 Open Problems and Future Work .................................. 230

Bibliography .......................................................... 233
Chapter 1

Introduction

Work on synchronization in shared memory multiprocessors has largely focused on the asynchronous model, either with or without crash failures. In wait-free synchronization, each process must make progress through its own steps regardless of the execution speeds or crash failures of others; in blocking synchronization, processes may busy-wait for others by repeatedly accessing shared memory, and so progress is guaranteed only when every active process is live. (A process is live if, whenever it begins executing an algorithm, it continues to take steps until the algorithm terminates.)

In this thesis we focus on blocking synchronization in asynchronous multiprocessors where processes communicate through shared memory. A natural way to measure the time complexity of algorithms in such multiprocessors is to count the number of memory accesses. This measure is problematic for blocking algorithms because, in this case, a process may perform an unbounded number of memory accesses while busy-waiting for another process. (For example, this happens in a mutual exclusion algorithm when a process waits for another to clear the critical section.) Instead, we can measure the time complexity of an algorithm by counting only remote memory references (RMRs), i.e., memory accesses that traverse the processor-to-memory interconnect. Local-spin algorithms, which perform busy-waiting by repeatedly reading shared variables that can be accessed locally, achieve bounded RMR complexity and have practical performance benefits [4].

The classification of memory accesses into local and remote depends on the memory model of the multiprocessor: In the distributed shared memory (DSM) model, a variable’s physical address determines locality with respect to a processor, each variable being local to exactly one processor and remote to all others. In the cache-coherent (CC) model,
processors operate on cached copies of shared variables, and it is the state of a processor’s local cache combined with the action of the coherence protocol (which keeps consistent copies of a variable in different caches) that determines locality. A memory access is local if it results in a cache hit and can be resolved without accessing main memory or a remote cache; a memory access is remote otherwise. To analyze the worst-case RMR complexity of an algorithm, we assume that each process runs on a distinct processor. (For this reason, we speak of processes and processors interchangeably.)

The main theme of this thesis is that certain popular synchronization primitives can be implemented efficiently in software, at least in terms of RMRs, from simpler ones. A synchronization primitive, in this context, is a type of operation that acts on a shared state abstractly represented by a memory word. We model an implementation of a set $S$ of primitives as a typed shared object that supports an atomic operation on its state corresponding to each primitive in $S$. Such an object supporting only read and write primitives is called a read/write register (or just register).

**Summary of results.**

(1) All comparison primitives [2], a class of synchronization primitives that includes the popular compare-and-swap (CAS) primitive, can be implemented using read and write operations with only a constant number of RMRs, in both the DSM and CC models. The same holds for the load-linked/store-conditional (LL/SC) pair of primitives. In both cases we show how to make the implementation readable and writable (i.e., we show how to support read and write primitives on the shared object).

(2) Our constant-RMR implementations can be made locally-accessible just like their hardware-implemented counterparts. In the DSM model, this means that the implemented object behaves as if it is local to some processor, and so some designated process can access the object without performing any RMRs. (This is nontrivial because the implemented object may use internally many base objects, not all local to the designated process.) Similarly, in the CC model the object behaves as if it can be cached, meaning that certain operations on an “in-cache” object cost no RMRs; whether an object is “in-cache” depends on the prior history of the execution and the coherence protocol. (This again is nontrivial because the implemented object may use internally many base objects and access them in complex ways.)

(3) As a consequence of (1) and (2), any CC or DSM shared memory algorithm using read, write, comparison primitives and LL/SC can be simulated by an algorithm that
uses only read and write operations, with only a constant-factor increase in the RMR complexity, while preserving other important properties.

Our constant-RMR implementations of comparison primitives and LL/SC are obtained in a series of steps. We first show how to transform any leader election algorithm that uses read and write operations into a name consensus algorithm that uses read and write operations and has the same worst-case RMR complexity to within a constant factor. (In a leader election algorithm, exactly one active process declares itself the winner, and all others declare themselves losers. In a name consensus algorithm, all active processes agree on one of their IDs.) Since there is a constant-RMR leader election algorithm [11], this transformation yields a constant-RMR name consensus algorithm. This efficient name consensus algorithm is used, in turn, to obtain constant-RMR CAS and LL/SC implementations from reads and writes. Finally, we observe that using CAS and no additional RMRs, one can easily implement any combination of comparison primitives.

**Related work and implications of our results.** Herlihy has shown that synchronization primitives vary widely in their ability to support wait-free implementations, and can be classified in the *wait-free hierarchy*, where the level of a primitive corresponds to its power [14]. For example, CAS together with read and write operations supports wait-free implementations of arbitrary objects shared by any number of processes; as a result, CAS is at the top level of the wait-free hierarchy. In contrast, the primitive *fetch-and-store* (FAS), together with read and write operations, supports wait-free implementation of arbitrary objects shared by at most two processes; as a result, FAS is at level two of the wait-free hierarchy.

As regards *blocking* synchronization, however, all primitives can be implemented using only read and write operations, by using such operations to implement mutual exclusion [9]. Thus, it is not meaningful to compare the power of two primitives by asking whether one can be used, along with read and write operations, to implement the other. Instead of comparing the power of two primitives based on computability, it is natural to ask whether we can base such a comparison on complexity — specifically, the RMR complexity of implementing a given primitive using read and write operations. From this point of view, we define the RMR complexity of a primitive, denoted $C$, as the minimum of worst-case RMR complexity over all implementations of that primitive using read and write operations; and we say that a primitive $S$ is stronger than a primitive $W$ if and
only if $\mathcal{C}(S) \in \omega(\mathcal{C}(W))$.

Looking at the relative power of primitives from this perspective reveals a landscape very different from that of Herlihy’s wait-free hierarchy [14]. Some primitives classified as strong in the wait-free hierarchy have low RMR-cost implementations from read and write operations, and are weak in their ability to solve mutual exclusion efficiently in terms of RMRs. Conversely, some primitives classified as weak in the wait-free hierarchy have inherently high RMR-cost implementations from reads and writes, and yield the most RMR-efficient mutual exclusion algorithms. For example, CAS is at the top of the wait-free hierarchy but, as we show in this thesis, it can be implemented from read and write operations using only a constant number of RMRs. On the other hand, FAS is only at level two of the wait-free hierarchy, but any implementation of FAS from read and write operations requires $\Omega(\log N)$ RMRs in the worst case (where $N$ is a parameter of the implementation that denotes the number of processes that can access it). This follows from the fact that mutual exclusion can be solved with $O(1)$ RMRs per passage through the critical section using FAS along with read and write operations [6], but requires $\Omega(\log N)$ RMRs per passage in the worst case using only reads and writes [5]. The same holds for the primitive *fetch-and-add* (FAA) [4], which is also weak in the wait-free hierarchy.

Anderson and Kim were the first to propose a way of ranking synchronization primitives according to their power for solving mutual exclusion efficiently (under the RMR complexity measure) [1], and to contrast this approach with Herlihy’s wait-free hierarchy [14]. To that end, they defined for each primitive (and value of $N$) a numerical rank $r$ that captures in a particular way the primitive’s ability to break symmetry among $N$ processes that apply it concurrently to the same variable. They then showed that a primitive of rank $r \geq 2$ can be used, in conjunction with read and write operations, to solve mutual exclusion using $O(\max(1, \log, N))$ RMRs per passage for any $N$. It is not known whether the latter bound is tight in general, although it is tight for common primitives such as reads and writes, CAS, FAS, and FAA.

Our ranking of synchronization primitives is similar to Anderson and Kim’s in that it captures, at least for common primitives, their power to efficiently solve mutual exclusion for any $N$. The advantage of our ranking scheme over Anderson and Kim’s is that it is based on a simpler property of “strength”, namely the RMR complexity of implementing a primitive using read and write operations. This property is easier to define than Anderson and Kim’s rank. Moreover, the RMR complexity bound obtained to evaluate
Chapter 1. Introduction

a primitive’s strength, in the sense we propose, is of independent interest.

Our results also have an interesting implication regarding mutual exclusion. To explain this we first recall certain facts about the RMR complexity of mutual exclusion. The most RMR-efficient mutual exclusion algorithm known to date that uses only read and write operations is one devised by Yang and Anderson; it performs $O(\log N)$ RMRs per passage through the critical section [26]. Attiya, Hendler and Woelfel showed that this is optimal [5], building on a prior $\Omega(\log N)$ lower bound by Fan and Lynch for a related but different cost model [10]. The optimality result holds for algorithms that use reads and writes only, and tightens a prior $\Omega(\log N/\log \log N)$ lower bound on RMRs by Anderson and Kim [2] that holds for a broader class of algorithms: those using reads, writes, and comparison primitives. Anderson and Kim posed the question whether $\Theta(\log N)$ is the tight worst-case RMR complexity lower bound for the latter class of algorithms [2]. We answer this in the affirmative through our result (3), in combination with the $\Omega(\log N)$ lower bound on RMR complexity of algorithms that use reads and writes only [5]. (For first-come-first-served (FCFS) mutual exclusion, the $\Omega(\log N)$ lower bound holds a fortiori, but the upper bound of $O(\log N)$ does not, because the simulation referred to by our result (3) breaks the FCFS property. The tight bound for FCFS mutual exclusion is established in [8].)

Road Map
We describe our model of computation in Chapter 2. Next, in Chapters 3–5 we present some building blocks needed for our main results. Chapter 6 then covers our implementations of CAS and LL/SC. Chapters 7 and 8 discuss how to make these implementations locally accessible and writable, respectively. In Chapter 9 we discuss implementing comparison primitives in general, and state our result (3). In Chapter 10 we give a universal construction of shared objects that are locally accessible in the DSM model. In Chapters 11 and 12 we bound the space complexity of our implementations. In Chapter 13 we conclude the thesis by discussing open problems.
Chapter 2

Model of Computation and Definitions

Our model of computation is based on Herlihy and Wing’s [15].

Processes and objects. There are $N$ asynchronous processes. Processes do not fail. The set of processes is denoted $\mathcal{P} = \{p_1, p_2, ..., p_N\}$, and we say that $p_i$ has $ID$ $i$. Processes communicate by applying operations on shared objects and receiving corresponding responses. Each process repeatedly applies such operations (one at a time) until it terminates, meaning that it has reached a special state where it remains indefinitely and takes no further action. A shared object represents a data structure with a well-defined set of states, as well as a set of operation types. The operation type determines the state transition that occurs when an operation of that type is applied to the shared object in a given state, as well as the response of the operation. It encodes the “signature” of the operation (including any arguments), as well as the ID of the process applying the operation. Processes and objects can be formally modelled as input/output automata [22], but here we adopt a more informal approach by describing the possible behaviours of processes and shared objects through pseudo-code. In pseudo-code, we refer to shared objects as variables.

Steps. A process interacts with shared objects by applying operations on these objects. We consider two types of operations: atomic and non-atomic. Atomic operations are instantaneous, and are represented as atomic steps. An atomic step where process $p$ applies an operation of type $ot$ to object $v$ and receives response $ret$ is denoted
Non-atomic operations are represented using separate invocation and response steps. An invocation step where process \( p \) invokes operation type \( ot \) on object \( v \) is denoted \((\text{INV}, p, v, ot)\). A response step where process \( p \) receives response \( ret \) from an operation execution on object \( v \) is denoted \((\text{RES}, p, v, ret)\). We say that a response step matches an invocation step if the two steps are applied by the same process to the same shared object.

**Histories.** A history \( H \) is a sequence of steps generated by processes. We explain how histories are generated later on and focus for now only on their building blocks. An operation execution in \( H \) is a pair consisting of an invocation step and the next matching response step, or just an invocation step if no matching response follows. We call an operation execution complete in the former case, and pending in the latter. Operation execution \( ox \) precedes operation execution \( ox' \) in \( H \) if the response of \( ox \) occurs before the invocation of \( ox' \) in \( H \). We say that \( ox \) and \( ox' \) are concurrent in \( H \) if neither precedes the other. A history \( H \) sequential if it only contains atomic steps, or if it only contains complete operation executions no two of which are concurrent. If \( H \) is sequential, then \( |H| \) denotes the number of atomic steps or operation executions in \( H \).

For any history \( H \) and set \( P \) of processes, we denote by \( H|P \) the subsequence of \( H \) consisting of all steps by processes in \( P \). Similarly, for any set \( V \) of shared objects, we denote by \( H|V \) the subsequence of \( H \) consisting of all steps on objects in \( V \). For a process \( p \) or object \( v \), we use \( H|p \) and \( H|v \) as shorthands for \( H|\{p\} \) and \( H|\{v\} \), respectively. We say that \( H \) is a history over \( V \) if \( H = H|V \).

For any history \( H \), we say that a process \( p \) is active in \( H \) if \( H|p \) is not empty. An infinite history \( H \) is fair if every process that is active in \( H \) either takes infinitely many steps or terminates.

**Object types and conformity to a type.** Every shared object has a type \( \tau = (S, s_{\text{init}}, O, R, \delta) \) where \( S \) is a set of states, \( s_{\text{init}} \in S \) is the initial state, \( O \) is a set of operation types, \( R \) is the set of responses, and \( \delta : S \times O \rightarrow S \times R \) is a (one-to-many) state transition mapping. The transition mapping \( \delta \) is intended to capture the behaviour of objects of type \( \tau \), in the absence of concurrency, as follows: if a process applies an operation of type \( ot \) to an object of type \( \tau \) that is in state \( s \), then the object may return to the process a response \( r \) and change its state to \( s' \) if and only if \((s', r) \in \delta(s, ot)\). An object \( v \) conforms to type \( \tau \) in a sequential history \( H \) if the steps in \( H|v \) are consistent with
some sequence of transitions of $\delta$ starting from state $s_{init}$, in the following sense: Letting $ot_i$ and $ret_i$ denote the operation type and response of the $i$'th atomic step or operation execution in $H|v$, and letting $k = |H|v|$, there exists a sequence $(s_0, s_1, s_2, \ldots, s_k)$ of states (in $S$) such that $s_0 = s_{init}$, and $(s_i, ret_i) \in \delta(s_{i-1}, ot_i)$ holds for all $i \leq k$.

**States.** Let $H$ be a sequential history over some set $B$ of objects, such that every object $v \in B$ conforms to its type in $H$. For any $k \leq |H|$, the state of the system (or simply “the state”) after $k$ atomic steps or operation executions in $H$ is denoted $H[k]$, and consists of the following: the state of each shared object and the private state of each process. The private state of a process comprises the values of private variables, in particular a “program counter” that determines the next pseudo-code statement executed by that process (and whether the process has terminated).

**Concurrent systems.** A concurrent system models algorithms where processes apply atomic steps on a set of shared objects. Formally, it is a tuple $S = (P, B, H)$ where $P$ is the set of processes, $B$ is the set of shared objects, and $H$ is the set of histories of the concurrent system. We will define $H$ informally through pseudo-code, consisting of one or more functions for each process, which are typically called according to some specific rules (e.g., each function is called at most once). The set $H$ then consists of histories generated by recording an atomic step for each access to a shared object incurred by processes as they execute their functions.

**Implementations of shared objects.** An implementation describes how to simulate a target object of a particular target type using a set of base objects of specified types. It is formally denoted as a tuple $I = (\tau, P, B, H)$ where $\tau$ is the target object type, $P$ is the set of processes, $B$ is the set of base objects, and $H$ is the set of histories. The histories in $H$ are over the base objects in $B$ and the target object of type $\tau$, denoted $O_{\tau}$. Informally, we describe an implementation using pseudo-code to define an access procedure for each operation type $ot$ of the target type and each process $p$. The pseudo-code for this access procedure describes how process $p$ applies an operation of type $ot$ to the target object, and computes the response of that operation, by applying operations on the base objects. The set $H$ consists of histories generated by recording an invocation or response step on the target object whenever a process begins or finishes executing an access procedure (respectively), and an atomic step for each access to a base object in $B$. 
Two correctness properties are required in every implementation: linearizability (safety) and termination (liveness).

**Linearizability and termination.**

Linearizability [15] is widely accepted as a correctness condition for histories of shared object implementations. Informally, it states that each operation execution on the target object appears to take effect instantaneously at some point between the operation execution’s invocation and response (or possibly not at all if the operation execution is pending). Formally, we first define for any history $H$ of an implementation $I = (\tau, P, B, H)$ a completion, which is a history $H'$ of invocation and response steps on the target object $O_\tau$ such that for every process $p$, $H'|O_\tau|p$ contains the same steps as $H|O_\tau|p$, except that for any operation execution $Op$ that is pending in $H$, either $Op$ is discarded from $H'$ or a matching response step follows the invocation step of $Op$ in $H'$. A history $H$ of an implementation $I = (\tau, P, B, H)$ is linearizable with respect to type $\tau$ if there exists a history $\bar{H}$ that satisfies the following properties:

(a) $\bar{H}$ is a sequential completion of $H$.

(b) The total order of operation executions in $\bar{H}$ is consistent with the partial order of the corresponding operation executions in $H$.

(c) The target object $O_\tau$ conforms to type $\tau$ in $\bar{H}$.

The termination property for a shared object implementation $I = (\tau, P, B, H)$ states that for any history $H$ of $I$, if $H$ is fair then every operation execution on the target object $O_\tau$ in $H$ is complete.

**Local and Remote Memory References.** In this thesis, we consider the cache-coherent (CC) and distributed shared memory (DSM) multiprocessor architectures, which are illustrated in Figure 2.1.

In each architecture, each memory access is either local or remote, as discussed briefly in Chapter 1. We now describe these concepts in detail in the context of accesses to the most fundamental of all shared objects: atomic read/write registers. In the DSM model, locality is defined statically: each object is local to exactly one process and is remote to all others, and so counting RMRs is straightforward. In the CC model, however, whether an access to an object is local or remote depends on the type of coherence protocol, and
the prior accesses to that object in the history under consideration. We consider two families of cache coherence protocols in this thesis: write-through and write-back [24].

In a write-through protocol, to read an object $v$ a process $p$ must have a (valid) cached copy of $v$. If it does, $p$ reads that copy without causing an RMR; otherwise, $p$ causes an RMR that creates a cached copy of $v$. To write $v$, $p$ causes an RMR that invalidates (i.e., effectively deletes) all other cached copies of $v$, and writes $v$ to memory (in the same RMR). (The write-through protocol comes in two flavours: with cache invalidation and with cache update upon write. In this thesis we consider only the invalidation version, as it is far more common in practice.)

In a write-back protocol, each cached copy is held in either “shared” or “exclusive” mode. To read an object $v$, a process $p$ must hold a cached copy of $v$ in either mode. If it does, $p$ reads that copy without causing an RMR. Otherwise, $p$ causes an RMR that: (a) eliminates any copy of $v$ held in exclusive mode, typically by downgrading the status of such a copy to shared and, if the exclusive copy was modified, writing that copy to memory; and (b) creates a local cached copy of $v$ held in shared mode. To write $v$, $p$ must have a cached copy of $v$ held in exclusive mode. If it does, $p$ writes that copy without causing RMRs. Otherwise, $p$ causes an RMR that: (a) invalidates all other cached copies of $v$ and writes any modified copy held in exclusive mode back to memory; and (b) creates a cached copy of $v$ held in exclusive mode.

In both protocols, a read of object $v$ by process $p$ causes an RMR if and only if $p$ has no (valid) cached copy of $v$. The protocols differ in the RMRs caused by writes: in write-through, every write causes an RMR; in write-back, a write of $v$ by $p$ causes an RMR if and only if $p$ does not hold a local cached copy of $v$ in exclusive mode.
It is possible to define RMRs precisely in the CC model under certain assumptions that capture “ideal” cache behaviour (e.g., ignoring RMRs due to finite cache size and false sharing). For our purposes, however, it suffices to define simple rules by which we can bound from above the number of RMRs incurred in a history. (We state these for sequential histories over atomic read/write registers here, and then generalize to other types of shared objects in Chapter 7.) For any history $H$ of atomic steps over a read/write register object $v$, and for any process $p$, each atomic step by $p$ causes an RMR, except in the situations described below.

In the write-through CC model:

(R) If $H'$ is a contiguous subsequence of $H$ where each atomic step is a read, then $p$’s atomic steps in $H'$ cause at most one RMR in total (to load $v$ into $p$’s cache).

In the write-back CC model, condition (R) holds, and furthermore:

(W) If $H'$ is a contiguous subsequence of $H$ where each atomic step is applied by $p$, then the atomic steps in $H'$ cause at most two RMRs in total (to load $v$ into $p$’s cache, and then possibly to promote $p$’s local copy of $v$ from shared to exclusive.)

Notation. We use the following notational conventions. In pseudo-code, $p_i$ denotes the process ID $i$, and $\text{PID}$ denotes the ID of the executing process. We denote by $\text{read}(\text{var})$ a read of shared variable $\text{var}$, returning the value read. Similarly, we denote by $\text{write} \ \text{var} := \text{val}$ a write of $\text{val}$ to shared variable $\text{var}$. We denote by $\text{await} \ \text{cond}$ a busy-wait loop that repeatedly evaluates condition $\text{cond}$, and terminates when $\text{cond}$ evaluates to true. The symbol $\triangleright$ in pseudo-code denotes access to a data structure field through a pointer, and is analogous to the operator $\rightarrow$ in C++ (e.g., $d \triangleright f$ denotes a field $f$ in a data structure pointed to by $d$). We use a variety of typefaces in pseudo-code to distinguish various programming constructs: reserved keyword, variable, FunctionName and constantName. Comments are formatted in C++ style.
Chapter 3

Consensus

In this chapter, we obtain an $O(1)$-RMR consensus algorithm for $N$ processes using reads and writes only. We consider the special case of consensus known as name consensus, from which ordinary consensus follows by a straightforward transformation that preserves RMR complexity, with only $O(1)$ additional RMRs per process.

Roughly speaking, in the name consensus (NC) problem the active processes must all agree on a common value, which is the name (ID) of one of them. A process wins if its name is agreed upon and loses otherwise. The problem is formally defined as follows.

First, since NC is a “one-shot” problem, processes must satisfy the following:

**Condition 3.1.** Each process calls `NameDecide()` at most once.

The correctness properties of name consensus are then defined as follows:

**Specification 3.2** (safety). For any history where Condition 3.1 holds:

(a) Each call to `NameDecide()` that terminates returns the ID of a process that has made a call to `NameDecide()`.

(b) No two calls to `NameDecide()` return different values.

**Specification 3.3** (liveness). For any fair history where Condition 3.1 holds, each call to `NameDecide()` terminates.

Name consensus is nontrivial in our model because the winner must be an active process, and not every process is required to be active; this rules out the naive algorithm that simply returns the ID of some fixed process.

We distinguish name consensus from the leader election (LE) problem, which was
Chapter 3. Consensus

13

solved with $O(1)$ RMRs using reads and writes in [11]. In the leader election problem, each active process executes a function $\text{LeaderElect}()$, which returns $\text{win}$ to exactly one process (the leader or winner), and $\text{lose}$ to all others. More formally, leader election is specified as follows:

**Condition 3.4.** Each process calls $\text{LeaderElect}()$ at most once.

**Specification 3.5** (safety). For any history where Condition 3.4 holds:

(a) If a call to $\text{LeaderElect}()$ terminates, then it returns either $\text{win}$ or $\text{lose}$.
(b) At most one call to $\text{LeaderElect}()$ returns $\text{win}$.
(c) If each call made to $\text{LeaderElect}()$ terminates, then exactly one such call returns $\text{win}$.

**Specification 3.6** (liveness). For any fair history where Condition 3.4 holds, each call to $\text{LeaderElect}()$ terminates.

Leader election is trivial to solve using name consensus (by comparing the winner’s ID to the caller’s ID). Similarly, in the CC model name consensus is easy to solve using leader election with only $O(1)$ additional RMRs per process. An algorithm that does this is presented in Figure 3.1. In this algorithm, processes first execute a leader election algorithm $L$ (line 1) and then either read or write a shared variable $\text{leader}$ initialized to $\bot$. The winner of $L$ writes its ID to $\text{leader}$ (line 2) and returns its own ID; other processes wait for $\text{leader} \neq \bot$ (line 4) and then return the value written to $\text{leader}$ by the winner. We will refer to the corresponding concurrent system (see Chapter 2) as $\mathcal{A}_{\text{NC-CC}}$. It is straightforward to show that $\mathcal{A}_{\text{NC-CC}}$ satisfies Specifications 3.2 and 3.3, and that $\text{Name Decide}()$ incurs only one more RMR than $L$ in the CC model. Thus, if we instantiate $L$ with a LE algorithm that uses only reads/writes and $O(1)$ RMRs per process in the CC model, such as the algorithm given in [11], we obtain a NC algorithm that uses only reads/writes and $O(1)$ RMRs per process in the CC model.

In the DSM model, the above algorithm is correct but has poor worst-case RMR complexity. This is because the variable $\text{leader}$ is local to exactly one process, and for all others the busy-wait loop at line 4 may generate an unbounded number of RMRs. (At line 4 a process reads $\text{leader}$ repeatedly until $\text{leader} \neq \bot$ holds, which may require an unbounded number of reads due to asynchrony.) In modifying this algorithm to achieve bounded RMR complexity in the DSM model, we must allow each process that does not
Chapter 3. Consensus

Declarations

**Shared variables:**

- leader - register, stores process ID or ⊥, initially ⊥

**Subroutines:**

- \( L \) - leader election algorithm

<table>
<thead>
<tr>
<th>Function NameDecide()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output:</strong> PID of winner</td>
</tr>
<tr>
<td>1 if ( L.\text{LeaderElect}() = \text{win} ) then</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3 else</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5 end</td>
</tr>
<tr>
<td>6 return read(leader)</td>
</tr>
</tbody>
</table>

Figure 3.1: Name consensus algorithm for the CC model.

Win \( L \) to busy-wait on its own locally-accessible shared variable until the winner is chosen. The winner must ensure that all of these variables are written, but it cannot write them itself using only \( O(1) \) RMRs because there may be up to \( N - 1 \) such variables, all remote to the winner. The technique presented in [11] for sharing work does not solve this problem because the winner does not know the IDs of all the processes that may be waiting for it (and hence are capable of sharing work). Nevertheless, a name consensus algorithm that builds on leader election is quite natural, and, as we show in the remainder of this chapter, is possible to construct with \( O(1) \)-RMR overhead in the DSM model using read and write operations. For the remainder of this chapter, we focus on the DSM model.

3.1 Name Consensus in the DSM Model: A High-Level Description

We now show how to solve name consensus at a cost of \( O(1) \) RMRs per process in the worst case in the DSM model. Our implementation uses as a building block an \( O(1) \)-RMR leader election algorithm \( L \) that uses only read/write registers, such as the one presented in [11], as well as some additional read/write registers. Our approach can also be used to construct a name consensus algorithm using any leader election algorithm, with at most a constant-factor increase in worst-case RMR complexity.

The way we use \( L \) is derived from the following simple observation: \textit{After any history of a leader election algorithm in which all active processes terminate, there is a “data flow” from the process elected leader to any other active process.} We can define the notion of data flow more precisely using graph-theoretic concepts. For each process \( p \), we first
define the following sets in the context of \( p \)'s execution of \( L \):

- \( W_p \) – set of processes to which the variables written remotely by \( p \) are local
- \( R_p \) – set of processes that wrote the values \( p \) read remotely

(N.B.: \( R_p \) is not necessarily the set of processes to which the variables read remotely by \( p \) are local. For example, if process \( p \) reads a remote variable \( v \) that is local to process \( q \), and the value that \( p \) reads in \( v \) was written by process \( r \), then \( R_p \) records \( r \) and not \( q \).)

Next, let \( G = (V, E) \) be the directed graph where \( V \) is the set of processes, and \((p, q) \in E\) if and only if \( q \in W_p \) (i.e., \( p \) wrote remotely to \( q \)'s memory) or \( p \in R_q \) (i.e., \( q \) read remotely a value written by \( p \)). If \( l \) is the process elected leader in \( L \), then a data flow from \( l \) to another process \( p \) is a path from \( l \) to \( p \) in \( G \). We will prove later (see Lemma 3.14) that there is such a path from \( l \) to every active process. Intuitively, this is because if such a data flow does not exist, we can construct a new execution of algorithm \( L \) in which only processes to which such paths do not exist are active, and they behave exactly as in the execution of \( L \) that gave rise to \( G \); these processes would elect \( l \), which is not active in this new execution.

The high-level idea behind the NC algorithm is as follows. The leader \( l \) elected in \( L \) writes its ID in a variable \textit{leader}, and will be the winner agreed on in NC. It then signals its out-neighbours in \( G \) to let them know that the winner’s name has been decided; once signalled, each neighbour repeats this step: it signals its out-neighbours; and so on. By our earlier observation, that there is a path from \( l \) to all active processes in \( G \), eventually all active processes will in fact be signalled, and each can simply read the winner’s name from \textit{leader}.

A number of issues must be addressed for this idea to work, and moreover to work with the required \( O(1) \)-RMR complexity per process:

(a) A process \( p \) does not always know all its out-neighbours: it knows the out-neighbours in \( W_p \), but does not necessarily know every \( q \) such that \( p \in R_q \).

(b) In fact, because of asynchrony, \( p \) might not be able to ever discover some of its out-neighbours. For example, suppose that \( p \) executes the NC algorithm (and within it \( L \)) and writes some register. After \( p \) terminates, \( q \) “wakes up” and reads a value written by \( p \) while executing \( L \), so that \( p \in R_q \). Thus, \( q \) is an out-neighbour of \( p \), and yet \( p \) has finished the NC algorithm and cannot be expected to signal \( q \).
(c) Even if a process $p$ knows all its out-neighbours, it cannot signal them by simply writing into their local memory (while they busy-wait) because there may be many processes $q$ such that $p \in R_q$. For example, suppose that in the LE portion of the NC algorithm, $p$ writes a value that is read by all other processes. Then $p$ has $N - 1$ out-neighbours, and so if $p$ had to write a variable local to each, it would be using $\Theta(N)$ RMRs instead of $O(1)$.

We address these issues as follows. To solve (a) and (b), we use an idea introduced in [11]: a “handshaking protocol” that allows a process $p$ to synchronize with each out-neighbour $q$ such that $p \in R_q$ by either discovering $q$’s ID, or letting $q$ know (in case $q$ becomes active much later than $p$) that $p$ is “out of the picture”. In the former case, $q$ waits for a signal from $p$. In the latter case, $q$ knows not to wait for $p$, and can read the winner’s ID from some shared variable (e.g., one written by $l$ before $p$ executes its side of the handshaking protocol). Thus, $p$ need not know all of its out-neighbours (which solves (a)), and “latecomers” can discover the winner’s ID easily (which solves (b)). Finally, for problem (c) we use a work sharing mechanism from [11] that spreads RMRs among the processes being signalled, namely the out-neighbours of $p$.

We now describe the building blocks of the name consensus algorithm in detail in Chapters 3.2–3.4, and then present the NC algorithm itself in Chapter 3.5.

### 3.2 Instrumented Leader Election Algorithm

In order to compute the sets $R$ and $W$ defined earlier, processes execute an “instrumented” version of $L$ (denoted $\hat{L}$) rather than using $L$ directly. This algorithm returns the same response as $L$, and also computes $R_p$ and $W_p$ for every active process $p$. More precisely, we construct $\hat{L}$ from $L$ as follows: For every register $r$ initialized to value $x$ by $L$, initialize $r$ to $(\perp, x)$. For every uninitialized register $r$ that may be accessed in $L$, initialize $r$ to $(\perp, \tilde{x})$ for some arbitrary value $\tilde{x}$. Each active process $p \in \mathcal{P}$ records $R_p, W_p \subseteq \mathcal{P}$ as private variables, both initialized to the empty set. To execute $\hat{L}.\text{LeaderElect}()$, a process $p$ begins by simulating its operations in $L.\text{LeaderElect}()$. If its next operation in $L.\text{LeaderElect}()$ writes value $x$ to register $r$ that is local to $q$, then $p$ writes $(p, x)$ to $r$, and adds $q$ to $W_p$ if $q \neq p$. Process $p$ also simulates the change in private state following its write of $r$. If its next operation in $L.\text{LeaderElect}()$ reads register $r$ that is local to process $q$, then $p$ reads $r$. If $(z, x)$ is the pair that $p$ read, then $p$ adds $z$
to $R_p$ if $z \neq \perp$, $z \neq p$, and $q \neq p$. Here $p$ also simulates the change in private state following its read of $r$, treating $x$ as the value read. Process $p$ repeatedly simulates its steps in $L$.$\text{LeaderElect}()$ in this manner until termination, at which time $p$’s execution of $\hat{L}$.$\text{LeaderElect}()$ produces the sets $R_p$ and $W_p$, and finally returns to $p$ the response of $L$.$\text{LeaderElect}()$.

They key properties of $\hat{L}$ are captured in the following lemma. Let $A_{\text{LE-I-DSM}}$ denote the corresponding concurrent system.

**Lemma 3.7.** For any history $H$ of $A_{\text{LE-I-DSM}}$ where Condition 3.4 holds:

(a) Specifications 3.5 and 3.6 hold.

(b) Each call to $\hat{L}$.$\text{LeaderElect}()$ incurs $O(1)$ RMRs in the DSM model.

(c) For each process $p$, the sets $R_p$ and $W_p$ generated during a call to $\hat{L}$.$\text{LeaderElect}()$ have size $O(1)$.

**Proof.** Let $L$ denote the $O(1)$-RMR leader election algorithm used to construct $\hat{L}$, and note that by our construction of $\hat{L}$, Condition 3.4 holds with respect to $\hat{L}$ in $H$. Furthermore, each process either receives the same response from $\hat{L}$.$\text{LeaderElect}()$ as from $L$.$\text{LeaderElect}()$ in $H$, or it does not complete its call to $\hat{L}$.$\text{LeaderElect}()$. Thus, if $H$ violates Specification 3.5 with respect to $\hat{L}$, then it does the same with respect to $L$. Similarly, if $H$ is fair and violates Specification 3.6 with respect to $\hat{L}$, then there must be a non-terminating call to $L$.$\text{LeaderElect}()$ in $H$, and so $H$ violates Specification 3.6 with respect to $L$. Since $L$ satisfies Specifications 3.5 and 3.6, this implies $\hat{L}$ does also, and so (a) holds.

For parts (b)–(c), recall that $\hat{L}$.$\text{LeaderElect}()$ simulates steps of $L$.$\text{LeaderElect}()$, performing an RMR at each step only if $L$.$\text{LeaderElect}()$ does so, and adding an element to either $R$ or $W$ only when an RMR occurs. Since $R$ and $W$ are private variables, part (b) follows from the $O(1)$ RMR complexity of $L$. Similarly, for any process $p$ since $R_p$ and $W_p$ are initially empty, $|R_p| + |W_p|$ is bounded from above by the RMR complexity of $L$, which implies part (c).

## 3.3 Handshaking Protocol

Our handshaking protocol is used by a process $p$ to synchronize with each out-neighbour $q$ such that $p \in R_q$. The protocol is similar in spirit to the one presented in [11], but somewhat simpler. For handshaking between $p$ and an out-neighbour $q \neq p$, the protocol relies
on a two-process $O(1)$-RMR leader election algorithm for $p$ and $q$. Such an algorithm must be accessed according to the following etiquette (in addition to Condition 3.4):

**Condition 3.8.** A two-process algorithm for processes $p$ and $q$ can only be accessed by $p$ and $q$.

The particular LE algorithm we use is *local to $p$* meaning that, in addition to Specifications 3.5 and 3.6, it satisfies the following:

**Specification 3.9.** A call to $\text{LeaderElect}()$ by process $p$ incurs zero RMRs.

An $O(1)$-RMR two-process LE algorithm satisfying this property is presented in detail in [11]. For completeness we repeat this algorithm in Figure 3.2. The algorithm uses two shared variables $A$ and $B$, both local to $p$. For this reason $p$’s ID must be fixed in advance. Process $p$ uses $A$ to record its progress, and similarly $q$ uses $B$. An interesting feature of the algorithm is that $q$’s execution path is wait-free, which means that its RMR complexity is $O(1)$ regardless of $q$’s ID (because $q$ never busy-waits). Process $q$ first assigns $B = 1$ (line 18), then checks $p$’s progress by reading $A$ (line 19), and decides on the basis of that whether it has won. If $p$ has begun its algorithm (i.e., $A \neq 0$) then $q$ assigns $B = 2$ and loses (lines 20–21). Otherwise $p$ has not begun its algorithm and so $q$ assigns $B = 3$ and wins (lines 23–24). Process $p$’s algorithm begins similarly, by assigning $A = 1$ (line 7) and reading $B$ (line 8). If $q$ has not begun its algorithm (i.e., $B = 0$) then $p$ wins (line 9). Otherwise, $p$ waits for $q$ to decide the outcome (i.e., assign $B \neq 1$) at line 11, then reads $B$ (line 12) to discover the decision, and finally returns *win* or *lose* accordingly.

The LE algorithm in Figure 3.2 is somewhat similar to Peterson’s two-process mutex [25] and to Lamport’s splitter [20] in the sense that it uses only a few atomic registers and each register takes on only a few possible values. There are also important differences. Peterson’s algorithm solves mutual exclusion and therefore cannot have bounded RMR complexity for one process and zero RMR complexity for the other process. Lamport’s splitter is wait-free and therefore cannot solve consensus for two processes [14], hence it cannot solve leader election for two processes either.

The LE algorithm is used for handshaking between $p$ and $q$ as follows: Process $p$ initiates an instance of the algorithm local to itself with each process $q \neq p$. Note that by Specification 3.9, $p$ incurs no RMRs in any of these. With respect to any particular process $q \neq p$, there are two outcomes of the LE algorithm for $p$ and $q$: If process $q$
Chapter 3. Consensus

Declarations

Shared variables:
- \( A \) – value in \{0, 1\}, initially 0, local to \( p \)
- \( B \) – value in \{0, 1, 2, 3\}, initially 0, local to \( p \)

Function \texttt{LeaderElect()} for \texttt{p}

Output: \texttt{win} or \texttt{lose}

7 \hspace{1em} \texttt{write} \( A := 1 \)
8 \hspace{1em} \textbf{if} \( \text{read}(B) = 0 \) \textbf{then}
9 \hspace{2em} \textbf{return} \texttt{win}
10 \hspace{1em} \textbf{else}
11 \hspace{2em} \textbf{await} \( B \neq 1 \)
12 \hspace{2em} \textbf{if} \( \text{read}(B) = 2 \) \textbf{then}
13 \hspace{3em} \textbf{return} \texttt{win}
14 \hspace{2em} \textbf{else}
15 \hspace{3em} \textbf{return} \texttt{lose}
16 \hspace{1em} \textbf{end}
17 \hspace{1em} \textbf{end}

Function \texttt{LeaderElect()} for \texttt{q}

Output: \texttt{win} or \texttt{lose}

18 \hspace{1em} \texttt{write} \( B := 1 \)
19 \hspace{1em} \textbf{if} \( \text{read}(A) = 1 \) \textbf{then}
20 \hspace{2em} \texttt{write} \( B := 2 \)
21 \hspace{2em} \textbf{return} \texttt{lose}
22 \hspace{1em} \textbf{else}
23 \hspace{2em} \texttt{write} \( B := 3 \)
24 \hspace{2em} \textbf{return} \texttt{win}
25 \hspace{1em} \textbf{end}

Figure 3.2: Two-process leader election algorithm for processes \( p \) and \( q \).

wins, then we say that \( q \) contacted \( p \), otherwise \( q \) failed to contact \( p \). (This is similar to terminology used in [11].) In the former case (\( q \) wins), \( p \) eventually loses, and knows that \( q \) is an active out-neighbour of \( p \) such that \( p \in R_q \). Thus, \( q \) can wait for a signal from \( p \), and \( p \) knows that it must signal \( q \). In the latter case (\( q \) loses), \( p \) eventually wins, and behaves as if \( q \) were not active. Thus, \( p \) does not signal \( q \), and hence \( q \) does not wait for a signal from \( p \).

Note that up to \( N - 1 \) processes may contact \( p \), and that \( p \) incurs zero RMRs handshaking with these processes, since the LE algorithm used is local to \( p \). Also, note that \( p \) may contact \( q \) even if \( q \) failed to contact \( p \), since there may be two “sessions” of the handshaking protocol between \( p \) and \( q \), running in “opposite directions.”

3.4 Signalling Mechanism

As mentioned before, our name consensus algorithm disseminates the leader’s ID across the data flow graph \( G \). The straightforward algorithm for doing this, whereby each process \( p \) signals all its out-neighbours in \( G \) (that it is aware of) by writing into their
local memory, is too expensive in terms of RMRs. Instead, we use a signalling mechanism that shares the workload among p’s neighbours.

Informally, the signalling mechanism works as follows. When p needs to communicate the leader’s ID to a subset \( \mathcal{N}_p \) of its neighbours (e.g., those which p has contacted), it builds a chain of the IDs from \( \mathcal{N}_p \) in its local memory. Process p then signals the first process in the chain, say q, by writing p’s ID in a designated location in q’s memory. This costs p a single RMR. Each process q that is signalled in this way then reads the leader’s ID from p’s memory and signals the next process in the chain (if any), whose ID it also reads from p’s local memory. The handshaking protocols executed prior to this ensure that for each such process p, all the processes in \( \mathcal{N}_p \) wait for p’s (either direct or indirect) signal, as otherwise the signalling mechanism breaks.

The signalling mechanism consists of subroutines signal, wait, and wait-any, which are presented in Figure 3.3. Function signal\( (P) \) tells the processes in set P that some event, for which they are waiting, has occurred. Function wait\( (q) \) blocks until a signal by process q occurs. Function wait-any blocks until a signal by any process occurs.

Function signal\( (P) \) is implemented as follows. At lines 27–31, the calling process p uses its local Work array to create a “chain” of identifiers from P (all elements of Work\[p\][1..N] are initially ⊥). This chain determines the order in which the processes in P signal each other. To bootstrap the signalling mechanism, p assigns true to D[q]\[p\] where q is the process at the beginning of the chain (line 32).

Function wait\( (q) \) is the counterpart of signal\( (P) \). The argument of wait\( (q) \) is the ID of a process q that may signal the caller. The process p executing wait waits for a signal from q by locally spinning on D[p]\[q\] (which is initially false) until a process that precedes it in a signalling chain writes true to D[p]\[q\] (see line 33). At line 34, p reads the identifier of the next process in the chain (if any). If such a process exists, then p signals it (lines 35–37).

We also define a function wait-any, which takes no argument, and terminates when the calling process has been signalled through any signalling chain. In contrast to wait, this function does not signal the next process in the chain; a process must call wait subsequently for this to happen.

The correctness properties of signal, wait and wait-any, are formally stated by the following lemma.

**Lemma 3.10 (safety).** For any history:
Chapter 3. Consensus

Declarations for signal, wait and wait-any.

Shared variables:
- $\text{Work}[1..N][1..N]$ – array of process ID or $\bot$, initially all $\bot$, elements
- $\text{Work}[p][1..N]$ local to process $p$
- $D[1..N][1..N]$ – array of Boolean, initially all $\text{false}$, elements $D[p][1..N]$ local to $p$

Private variables: (per-process)
- $\text{prev}, \text{next}, t$ – process ID or $\bot$, uninitialized

Function $\text{signal}(P)$

Input: $P \subseteq \mathcal{P}$

if $P = \emptyset$ then return

// Create a ‘‘signalling chain’’ from elements of $P$.

$\text{prev} := \bot$

foreach next $\in P$ do

// Order of chain elements is reverse of loop order.

write $\text{Work}[\text{PID}][\text{next}] := \text{prev}$

$\text{prev} := \text{next}$
end

// Signal the first process in the chain.

write $D[\text{next}][\text{PID}] := \text{true}$

Function $\text{wait}(q)$

Input: $q \in \mathcal{P}$

await $D[\text{PID}][q] = \text{true}$

// Identify next process in the signalling chain.

$\text{next} := \text{read}(\text{Work}[q][\text{PID}])$

if $\text{next} \neq \bot$ then

// Signal the next process.

write $D[\text{next}][q] := \text{true}$
end

Function $\text{wait-any}()$

loop forever

// Wait for signal from any process.

foreach $t \in \mathcal{P}$ do

if $D[\text{PID}][t] = \text{true}$ then

return
end
end

end

Figure 3.3: Work-sharing signalling mechanism.
(a) In the DSM model, each call to \texttt{signal} and \texttt{wait} incurs $O(1)$ RMRs, and each call to \texttt{wait-any} incurs zero RMRs.

(b) Each call to \texttt{signal} performs a bounded number of steps.

(c) If process $p$ completes a call to \texttt{wait}(q) then $q$ previously made a call to \texttt{signal}(P) with $p \in P$.

(d) If process $p$ completes a call to \texttt{wait-any()} then some process $q$ previously made a call to \texttt{signal}(P) with $p \in P$.

Proof.

Parts (a) and (b): These follow directly from the algorithms and the locality of $\text{Work}[p][1..N]$ to process $p$.

Part (c): Suppose that $p$ completes a call to \texttt{wait}(q). Then $p$ reads $D[p][q] = \text{true}$ at line 33. Since this variable is initially \texttt{false}, some process $r$ must have assigned $D[p][q] = \text{true}$, either at line 32 of \texttt{signal}, or at line 36 of \texttt{wait}. In the former case, it follows from the algorithm for \texttt{signal}(P) that $r = q$ and that $p \in P$. In the latter case, $r$ read $q$’s ID from $\text{Work}[q][r]$ at line 34, which must have been written at line 29 of \texttt{signal}(P), namely by $q$ with $p \in P$.

Part (d): The proof is a simplified version of the proof of part (c).

Lemma 3.11 (liveness). For any fair history:

(a) If some process $q$ writes \texttt{true} to $D[p][q]$ at line 32 or at line 36, and if $p$ calls \texttt{wait}(q), then $p$’s call terminates.

(b) If some process $q$ writes \texttt{true} to $D[p][q]$ at line 32 or at line 36, and if $p$ calls \texttt{wait-any}, then $p$’s call terminates.

Proof. Part (a): Follows directly from the fact that once a process assigns $D[p][q] = \text{true}$, this variable is never reset back to \texttt{false}. Thus, any call to \texttt{wait}(q) by $p$ in a fair history eventually progresses beyond line 33 and terminates.

Part (b): The proof is analogous to the proof of part (a).

Lemma 3.12 (liveness). Consider any fair history $H$ where processes call the subroutines \texttt{signal}, \texttt{wait} and \texttt{wait-any}. Consider a particular call to \texttt{signal}(P) in $H$, say by some process $q$. Suppose that no other call by $q$ to \texttt{signal}(P) occurs with $P \cap P' \neq \emptyset$. Also suppose that in $H$ every process $p \in P$ either makes a call to \texttt{wait}(q) or makes a non-terminating call to \texttt{wait-any}. Then all the calls to \texttt{wait}(q) and \texttt{wait-any} made in $H$ by processes in $P$ terminate.
Proof. If \(|P| = 0\) then the result follows trivially, so consider the case when \(|P| \geq 1\). Let \(\sigma = \langle p_1, p_2, ..., p_m \rangle\) be the sequence of process IDs selected at line 28 during \(q\)'s execution of \(\text{signal}(P)\) under consideration, in reverse order (i.e., \(q\) assigns \(D[p_1][q] = \text{true}\) at line 32). Note that \(\sigma\) is a sequence over all the elements of \(P\) without repetition. Let \(S(k)\) represent the following claim: all calls to \text{wait-any} and \text{wait} under consideration made by the first \(k\) processes in \(\sigma\) terminate. We will show that \(S(k)\) holds for all \(k \in [1..m]\) by induction on \(k\), which implies the lemma.

Basis: \(S(1)\) follows from Lemma 3.11 (a) and (b) since \(q\) assigns \(D[p_1][q] = \text{true}\) at line 32 during its call to \text{signal}(P).

Induction step: Suppose that \(m \geq 2\) (i.e., \(|P| \geq 2\)). For any \(k \in [2..m]\), and for all \(i \in [0..k - 1]\), suppose that \(S(i)\) holds. We must show that \(S(k)\) holds. By the induction hypothesis, it suffices to show that the calls to \text{wait-any} and \text{wait}(q) by \(p_k\) terminate. To that end, we will show that \(p_k\) assigns \(D[p_k][q] = \text{true}\) at line 36 of \text{wait}(q). To see this, first note that by the induction hypothesis and our assumption on when processes call \text{wait}(q), \(p_{k-1}\) eventually calls \text{wait}(q) and reads \(\text{Work}[q][\text{PID}]\) at line 34. Since we assume \(q\) does not call \text{signal}(P') with \(P \cap P' \neq \emptyset\), it follows that \(p_{k-1}\) reads \(p_k\)'s ID from \(\text{Work}[q][\text{PID}]\), and then assigns \(D[p_k][q] = \text{true}\) at line 36. Consequently, any call to \text{wait-any} by \(p_k\) terminates by Lemma 3.11 (b) and any call to \text{wait}(q) by \(p_k\) terminates by Lemma 3.11 (a), as wanted. 

\[
\square
\]

3.5 Name Consensus in the DSM Model: A Detailed Description

The name consensus algorithm (\text{NameDecide()} ) that uses the instrumented leader election algorithm \(\hat{L}\), handshaking protocol, and signalling mechanism described earlier, is presented in Figures 3.4–3.5. We will refer to the corresponding concurrent system (see Chapter 2) as \(A_{\text{NC-DSM}}\). The algorithm uses \(N^2\) “instances” of a two-process leader election algorithm for handshaking, each instance having its own distinct copy of the underlying shared variables. We denote the instance for \(p\) and \(q\), which is local to \(p\), by \(L2P[p][q]\).

Leader election using \(\hat{L}\) occurs at line 45. For each process \(p\), the computation of the sets \(R_p\) and \(W_p\) is performed implicitly by \(\hat{L}\). We refer to \(p\) and \(q\) as \textit{neighbours} if and only if \(p\) and \(q\) are adjacent (ignoring direction of edges) in the (directed) “data flow”
Declarations

**Shared variables:** (global)
- $\hat{L}$ – instrumented $O(1)$-RMR leader election algorithm (see Section 3.2)
- $leader$ – process ID or $\perp$, initially $\perp$
- $L2P[1..N][1..N]$ – array of $O(1)$-RMR two-process LE algorithms, where $L2P[p][q]$ is for $p$ and $q$, and is local to $p$

**Private variables:** (per-process)
- $R, W, U, X$ – sets of process IDs, initially $\emptyset$
- $q$ – process ID, uninitialized

Figure 3.4: Name consensus algorithm for the DSM model – part 1 (declarations).

graph $G$ defined in Section 3.1 (i.e., $p \in R_q \cup W_q$ or $q \in R_p \cup W_p$). As we prove later, the graph $G$ has useful connectedness properties.

After computing $R_p$ and $W_p$, process $p$ needs to communicate with its neighbours regarding the leader’s ID, which is the response (i.e., winner) of the name consensus algorithm. As explained in Section 3.1, the high-level idea is to propagate this response through $G$, along the directed edges and away from the leader. We refer to this informally as propagating information “downstream” in $G$, even though $G$ may contain cycles. The leader’s ID is itself stored in a global shared register $leader$, initially $\perp$. If $p$ is not the leader, then it attempts to discover the leader’s ID using two mechanisms. First, it attempts to “pull” information from an upstream neighbour on a directed path from the leader to $p$. To that end, $p$ engages in the handshaking protocol described in Section 3.3 and tries to contact every neighbour in $R_p$. If $p$ fails to contact some such neighbour then that neighbour already knows the leader’s ID and so $p$ can read this ID immediately from $leader$. On the other hand, if $p$ succeeds in contacting each neighbour then none of these neighbours knows the leader’s ID and so $p$ waits for some neighbour (not necessarily one it has contacted) to “push” information to it. More precisely, “push” means that some neighbour signals $p$ and then $p$ reads $leader$. Finally, once $p$ discovers the leader’s ID (through either the pull or push mechanism), it pushes information to all its neighbours in $W_p$, and to any neighbour that contacted $p$ using the handshaking protocol. For subtle reasons related to the work-sharing signalling mechanism, $p$ may have to perform additional work at this point to ensure that processes it contacted earlier make progress.

Now consider the outcome of executing line 45. If $p$ is elected leader, then it writes its ID to $leader$ at line 46. At lines 48–52, $p$ tries to contact its neighbours from $R_p$. 

---

Chapter 3. Consensus 24
Function NameDecide()

Output: PID of leader

// Execute "instrumented" leader election algorithm.

45 if $\hat{L}.\text{LeaderElect}() = \text{win}$ then
46    write $leader := \text{PID}$
47 end

// Note: sets $R$ and $W$ have been computed (implicitly) at line 46.
// Try to contact neighbours in $R$.
48 foreach $q \in R$ do
49    if $L2P[q][\text{PID}].\text{LeaderElect}() = \text{win}$ then
50        $U := U \cup \{q\}$
51    end
52 end

// If needed, wait for a signal (that $leader \neq \bot$).
53 if read($leader$) = $\bot$ then
54    wait-any()
55 end

// Invariant: $leader \neq \bot$. Next, signal out-neighbours in $W$.
56 foreach $q \in W$ do
57    signal($\{q\}$)
58 end

// Discover other out-neighbours.
59 foreach $q \in \mathcal{P} \setminus \{\text{PID}\}$ do
60    if $L2P[\text{PID}][q].\text{LeaderElect}() = \text{lose}$ then
61        $X := X \cup \{q\}$
62    end
63 end

// Signal discovered neighbours except those already signalled.
64 signal($X \setminus W$)

// Share work in signalling chains of neighbours contacted at line 49.
65 foreach $q \in U$ do in parallel
66    wait($q$) // Note: loop body executed concurrently for all $q$.
67 end parallel

// All parallel calls to wait at line 66 completed by now. Ready to return.
68 return read($leader$)

Figure 3.5: Name consensus algorithm for the DSM model – part 2 (function NameDecide).
(Since $R_p$ has $O(1)$ elements (see Lemma 3.7 (c)), this takes $O(1)$ RMRs in total.) Here $p$ stores in the set $U$ the IDs of neighbours that were actually contacted. If there is some $q \in R_p$ that $p$ fails to contact, then $q$ has progressed to line 60, and so as we show in Lemma 3.15, $\text{leader} \neq \bot$. If $p$ does not win at line 45 and it contacts each $q \in R_p$, then $p$ still does not know the leader when it reaches line 53. In this case, $p$ waits at line 54 for a signal from any neighbour by calling \texttt{wait-any}().

By the time $p$ reaches line 56, $\text{leader} \neq \bot$ holds, as we show later in Lemma 3.15. Now $p$ must signal some of its neighbours of this. The algorithm deals first with $p$’s downstream neighbours in $W_p$. To that end, $p$ calls \texttt{signal}($\{q\}$) for each $q \in W_p$ at line 57. Next, $p$ attempts to communicate with other neighbours downstream of it in $G$, namely any process $q$ for which $p \in R_q$ holds, which means that $q$ tried to contact $p$ at line 49. If $q$ did contact $p$, then $q$ may rely on $p$ to signal it when $\text{leader} \neq \bot$. Consequently, $p$ determines the IDs of all such processes at lines 59–63, collects these IDs in the set $X$, and then signals them all at line 64. Processes participating in this signalling chain share work, which is necessary since there may be so many of them that $p$ cannot directly communicate with all of them in a constant number of RMRs.

The argument in $p$’s call to \texttt{signal} at line 64 is actually the set difference $X \setminus W$ and not $X$ itself, which is done for two reasons: First, is efficiency – since $p$ already called \texttt{signal}($\{q\}$) for each process $q \in W$ at line 57, it is not necessary to signal these processes again. The second reason is to meet the conditions of Lemma 3.12 – process $p$ may not call \texttt{signal} twice with arguments that are non-disjoint sets, as this may break the two signalling chains.

Finally, $p$ does its share of the work for each of the signalling chains it entered by contacting a neighbour at line 49. Each of these neighbours will try to signal $p$ that $\text{leader} \neq \bot$ by calling (via \texttt{signal}), and will rely on $p$ calling \texttt{wait} to assist in the signalling mechanism. Thus, $p$ calls \texttt{wait}($q$) for each $q \in U$ at line 66. More precisely, $p$ executes a parallel for loop at lines 65–67, which makes multiple calls to \texttt{wait} in parallel by interleaving the corresponding operations, say in round-robin fashion. (We introduce parallelism here only to facilitate exposition. We could equally well use a modified version of \texttt{wait} that waits on multiple processes, but such a subroutine is somewhat more difficult to specify and analyze.) Note that $p$ cannot wait for each $q \in U$ sequentially because if $p$ were to block inside a particular call to \texttt{wait}($q$), $p$ could prevent progress in the signalling chain corresponding to some other process in $U$, leading to deadlock. If $p$ reaches line 68, then all the parallel calls have terminated, and $\text{leader} \neq \bot$ holds (and
has held since \( p \) reached line 56). Thus, \( \text{NameDecide()} \) returns the leader’s ID to \( p \).

To prove the correctness of \( \text{NameDecide()} \) (see Theorem 3.19 at the end of this chapter), we first establish some technical lemmas.

**Lemma 3.13.** Let \( H \) be any history of \( A_{\text{NC-DSM}} \) in which Condition 3.1 holds, and every active process has completed the call to \( \hat{L}.\text{LeaderElect()} \) at line 45. For any pair of distinct processes \( p, q \in \mathcal{P} \), if \( p \) reads a value written by \( q \) while calling \( \hat{L}.\text{LeaderElect()} \) in \( H \), then once \( p \) completes this call, \( q \in R_p \) or \( p \in W_q \).

*Proof.* If, while executing line 45, \( p \) reads remotely a value written by \( q \), then \( p \) adds \( q \) to \( R_p \). Otherwise, \( p \) reads a value written by \( q \) in \( p \)’s local memory, in which case \( q \) adds \( p \) to \( W_q \), because \( p \) and \( q \) are distinct. \( \Box \)

**Lemma 3.14.** Let \( H \) be any history of \( A_{\text{NC-DSM}} \) where Condition 3.1 holds, and every active process has completed the call to \( \hat{L}.\text{LeaderElect()} \) at line 45. Suppose that the call to \( \hat{L}.\text{LeaderElect()} \) that ends last does so in step \( i \) of \( H \). Let \( G_H \) denote the directed graph \((V, E)\) where \( V \) is the set of process IDs and for any \( p, q \in V \), \((p, q) \in E\) iff \( p \in R_q \) or \( q \in W_p \) in state \( H[i] \) (and thereafter). Let \( l \) denote the process elected leader at line 45. Then, every process \( p \) active in \( H \) is reachable from \( l \) in \( G_H \).

*Proof.* Suppose, by way of contradiction, that there is some process \( p \) active in \( H \) that is not reachable from \( l \) in \( G_H \). Let \( R \) be the set of vertices of \( G_H \) that are reachable from \( l \), and let \( \bar{R} \) be the remaining vertices. Note that \( l \in R \) and \( p \in \bar{R} \), so both sets are nonempty. Our key observation is that \( H|\bar{R} \), like \( H \), is a history where each process in \( \bar{R} \) calls \( \text{NameDecide()} \) at most once. This is because by Lemma 3.13 and our definition of \( \bar{R} \), no process \( q \in \bar{R} \) reads (in \( H \) or in \( H|\bar{R} \)) from a shared variable a value written by a process in \( R \). (If this were false, there would be an edge in \( G_H \) from a vertex in \( R \) to a vertex in \( \bar{R} \), contradicting our definition of these sets.) Moreover, \( H|\bar{R} \) is a history of \( \text{NameDecide()} \) in which every active process completes its call to \( \hat{L}.\text{LeaderElect()} \) at line 45, and receives the same response as in \( H \). It follows from Specification 3.5 and Lemma 3.7 (a) that this response in \( H \) is lose for every process different from \( l \), which contradicts Lemma 3.7 (a) for \( H|\bar{R} \) because \( l \) is not active in \( H|\bar{R} \) by our definition of \( \bar{R} \). (Since \( l \) is reachable from itself, it belongs in \( R \) and not \( \bar{R} \).) \( \Box \)

**Lemma 3.15.** Let \( H \) be any history of \( A_{\text{NC-DSM}} \) where Condition 3.1 holds. If some process has reached line 56 in some prefix \( H' \) of \( H \), then \( \text{leader} \neq \bot \) holds in \( H \) after the prefix \( H' \).
Proof. First, note that \textit{leader} is written at most once during \( H \), namely at line 46 by the process elected \( \hat{L} \). Consequently, the property \( \text{leader} \neq \bot \) is stable in \( H \). It remains to show that for any process \( p \), if \( p \) reaches line 56 then \( \text{leader} \neq \bot \). Suppose, for contradiction, that this is false; without loss of generality let \( p \) be the first process to reach line 56 while \( \text{leader} = \bot \) in \( H \). A fortiori, \( \text{leader} = \bot \) when \( p \) reached line 53, and therefore \( p \) completed a call to \( \text{wait-any}(\cdot) \) at line 54. By Lemma 3.10 (d), some process \( q \) previously called \( \text{signal}(P) \) with \( p \in P \). Since \( q \) calls \( \text{signal} \) only after it reaches line 56, it follows that \( q \) reached line 56 before \( p \), and therefore while \( \text{leader} = \bot \). This contradicts the definition of \( p \) as the first process to reach line 56 while \( \text{leader} = \bot \).

\textbf{Lemma 3.16.} Let \( H \) be any history of \( A_{NC-DSM} \) where Condition 3.1 holds. The following hold in \( H \):

(a) For any processes \( p \) and \( q \), the \( \text{LE algorithm} L2P[p][q] \) is accessed according to Condition 3.4 and Condition 3.8.

(b) If \( H \) is fair, all executions of \( \text{signal} \) at line 64 satisfy all the hypotheses of Lemma 3.12.

\textbf{Proof. Part (a):} It suffices to show that the two-process \( \text{LE algorithm} L2P[p][q] \) is executed at most once by \( p \), at most once by \( q \), and only with \( p \neq q \). The only two places where \( L2P[p][q] \) is executed are line 49 and line 60. Thus, it follows from the loop conditions at line 48 and line 59 that if \( L2P[p][q] \) is accessed, then \( p \neq q \). (At line 49, this is because \( z \not\in R_z \) for any process \( z \), by construction of the set \( R \).) It also follows from the loops containing line line 49 and line 60 that only \( p \) and \( q \) execute \( L2P[p][q] \), at most once at each line. If \( p \) or \( q \) executes \( L2P[p][q] \) at both lines, then this implies \( p = q \), which contradicts our earlier observation.

\textbf{Part (b):} Consider a call to \( \text{signal}(P) \) made by some process \( q \) at line 64. To satisfy the hypotheses of Lemma 3.12, we must show two things: (1) \( q \) does not make another call to \( \text{signal}(P') \), where \( P \cap P' \neq \emptyset \); and (2) every process \( p \in P \) eventually makes a call to \( \text{wait}(q) \) at line 66, provided that any prior call it makes to \( \text{wait-any} \) at line 54 terminates. For (1), note that any other such call to \( \text{signal}(P') \) by \( q \) must have been at line 57, and in that case \( P' \) is a singleton set containing an element of \( W_q \). Then it follows from the set subtraction \( X \setminus W \) at line 64 that \( P \cap P' = \emptyset \), as wanted. For (2), consider any \( p \in P \). Note that since \( p \in P \) and by part (a) of this lemma, \( p \) must have contacted \( q \) at line 49 by Specification 3.5 and the algorithm. Thus, \( p \) is active. Furthermore, since
$H$ is fair, it follows from Lemma 3.16 (a), Specification 3.6, Lemma 3.10 (b), and the algorithm that $p$ either makes a non-terminating call to \texttt{wait-any} at line 54, or reaches line 65. In the latter case, $p$ begins the for loop at lines 65–67, where it makes a call to \texttt{wait($z$)} for each $z \in U_p$. Since these calls are made in parallel, and since $q \in U_p$ (because $p$ contacted $q$ at line 49), $p$ eventually makes a call to \texttt{wait($q$)}, as wanted. (Note that if a non-parallel for loop was used, $p$ might become blocked in some other call to \texttt{wait} before calling \texttt{wait($q$)}.)

\begin{lemma}
For any history $H$ of $A_{NC-DSM}$ where Condition 3.1 holds, each call to \texttt{NameDecide()} terminates.
\end{lemma}

\begin{proof}
Suppose, for contradiction, that \texttt{NameDecide()} does not terminate for a subset $B$ of processes that are active in $H$. By Lemma 3.7 (a), Specification 3.6, and fairness of $H$, every process active in $H$ eventually completes its call to \texttt{LeaderElect()} at line 45. Let $G_H$ be the directed graph corresponding to $H$, as defined in the statement of Lemma 3.14. Let $l$ be the process that is elected leader at line 45 during $H$. By Lemma 3.14, there is a directed path in $G_H$ from $l$ to every process in $B$.

Now choose any process $p \in B$. Since $H$ is fair, it follows from the structure of \texttt{NameDecide()} that $p$ makes progress until it makes a non-terminating call to \texttt{LeaderElect}, \texttt{signal}, \texttt{wait-any}, or \texttt{wait}. Every active process completes line 45, as argued earlier. All executions of the two-process leader election algorithm also terminate by Lemma 3.16 (a) and Specification 3.6. All executions of \texttt{signal} terminate by Lemma 3.10 (b). Thus, it remains to rule out the following two cases:

\textbf{Case A:} $p$ makes a non-terminating call to \texttt{wait-any} at line 54. Since $p$ reached line 54, it follows from the algorithm, particularly lines 45–46 and the test at line 53, that $p \neq l$. Now without loss of generality, assume that $p$ is chosen so that the length of the path from $l$ to $p$ in $G_H$ is minimal, and let $z$ be $p$’s upstream neighbour on this minimal path. Since there is an edge from $z$ to $p$ in this path, $z$ is active by our definition of $G_H$ (i.e., either $z$ wrote a variable local to $p$, or $p$ read a remote variable written by $z$). Moreover, $z \in R_p$ or $p \in W_z$.

\textbf{Subcase A-i:} $z \in R_p$. First, we will show that $p$ contacted $z$ at line 49. Suppose otherwise. Then by the time $p$ evaluates the condition at line 53, $z$ has already reached line 60 by Lemma 3.16 (a) and Specification 3.5. Consequently, Lemma 3.15 implies that $leader \neq \bot$ holds when $p$ is at line 53. But this contradicts $p$ branching to line 54 as stated in Case A. Thus, $p$ contacts $z$, which implies (by Lemma 3.16 (a) and Speci-
(at line 61) Consequently, \( z \) adds \( p \) to \( X_z \) at line 61, and then calls \( \text{signal}(X_z \setminus W_z) \) at line 64, where \( p \in X_z \setminus W_z \) unless \( z \) already called \( \text{signal}([p]) \) at line 57. In either case, \( p \)'s call to \text{wait-any} at line 54 terminates by Lemma 3.12 and Lemma 3.16 (b), which contradicts the hypothesis of Case A.

**Subcase A-ii:** \( p \in W_z \). Since \( z \) is active, and since it terminates in \( H \) by our selection of \( p \), it follows that \( z \) calls \( \text{signal}([p]) \) at line 57. In that case, \( p \)'s execution of \text{wait-any} at line 54 terminates by Lemma 3.11 (b), which contradicts the hypothesis of Case A.

**Case B:** \( p \) makes a non-terminating call to \text{wait}(q) at line 66 for some process \( q \). Note that \( p \) contacted \( q \) at line 49 since \( q \in U_p \) when \( p \) is at line 65. Now consider \( q \). Since \( q \in R_p \), \( p \) read a value written by \( q \) in \( H \), and so \( q \) is active in \( H \). Furthermore, \( q \neq p \), and by our prior analysis (up to and including Case A), \( q \) eventually reaches line 65. Since \( p \) contacted \( q \) and \( p \neq q \), it follows from Lemma 3.16 (a) and Specification 3.5 that \( q \) loses \( L_2P[z][p] \). \( \text{LeaderElect()} \) at line 60 and so \( p \in X_q \) when \( q \) reaches line 65. Consequently, either \( p \in W_q \) and \( q \) completed a call to \( \text{signal}([p]) \) at line 57, or \( p \notin W_q \) and \( q \) completed a call to \( \text{signal}(X_q \setminus W_q) \) at line 64 with \( p \in X_q \setminus W_q \). In either case, \( p \)'s call to \text{wait}(q) at line 66 terminates by Lemma 3.12 and Lemma 3.16 (b), which contradicts the hypothesis of Case B.

**Lemma 3.18.** For any history \( H \) of \( \mathcal{A}_{NC-DSM} \) where Condition 3.1 holds, each call to \( \text{NameDecide()} \) incurs \( O(1) \) RMRs in the DSM model.

**Proof.** Let \( p \) be any process that calls \( \text{NameDecide()} \). We will show that \( p \) performs \( O(1) \) RMRs in this call. We consider each line of \( \text{NameDecide()} \) where \( p \) may incur one or more RMRs, and argue that the number of RMRs \( p \) incurs at each such line is \( O(1) \).

- **lines 45–47:** The call to \( \hat{L}.\text{LeaderElect()} \) incurs \( O(1) \) RMRs by Lemma 3.7 (b). At most one additional RMR occurs at line 46.

- **lines 48–52:** The loop here has \( |R_p| \) iterations, where \( |R_p| \in O(1) \) by Lemma 3.7 (c). RMRs may occur only at line 49, and each execution of this line incurs \( O(1) \) RMRs by the RMR complexity of the two-process leader election algorithm, and by Lemma 3.16 (a).

- **lines 53–55:** There is at most one RMR at line 53, and the call to \text{wait-any} at line 55 incurs zero RMRs by Lemma 3.10 (a).
• lines 56–58: Each of these lines is executed \(|W_p|\) times, where \(|W_p| \in O(1)\) by Lemma 3.7 (c). Here RMRs occur only at line 57, and each execution of \texttt{signal} incurs \(O(1)\) RMRs by Lemma 3.10 (a).

• lines 59–63: The loop here has \(N\) iterations. RMRs may occur only at line 60, and in fact an execution of \(L2P[p][q].\text{LeaderElect}()\) by any process \(p\), for any \(q\), incurs zero RMRs by Specification 3.9 and Lemma 3.16 (a).

• line 64: The call to \texttt{signal} here incurs \(O(1)\) RMRs by Lemma 3.10 (a).

• lines 65–67: Each of these lines is executed \(|U_p|\) times, which is \(O(1)\) since \(U_p \subseteq R_p\) by the algorithm, and since \(|R_p| \in O(1)\) by Lemma 3.7 (c). Here RMRs occur only at line 66, where each execution of \texttt{wait} incurs \(O(1)\) RMRs by Lemma 3.10 (a).

• line 68: At most one RMR occurs here.

\[\text{Theorem 3.19.} \text{ For any history } H \text{ of } A_{NC-DSM} \text{ (Figures 3.4–3.5) where Condition 3.1 holds, } H \text{ satisfies Specifications 3.2 and 3.3. Furthermore, each call to NameDecide() in } H \text{ incurs } O(1) \text{ RMRs in the DSM model.}\]

\[\text{Proof.} \text{ Consider any history } H \text{ of } A_{NC-DSM} \text{ where Condition 3.1 holds.}\]

\[\text{Specification 3.2:} \text{ Suppose that each call to NameDecide() terminates in } H. \text{ It follows from the algorithm and Lemma 3.15 that NameDecide() returns to each caller the ID of the process } l \text{ that wins } \hat{L} \text{ at line 45. By Lemma 3.7 (a), } l \text{ is the ID of a process that called } \hat{L}.\text{LeaderElect}() \text{ in } H, \text{ hence also called NameDecide()}, \text{ as wanted.}\]

\[\text{Specification 3.3:} \text{ Suppose that } H \text{ is fair. Then each call to NameDecide() in } H \text{ terminates by Lemma 3.17.}\]

\[\text{RMR complexity:} \text{ This follows directly from Lemma 3.18.}\]

3.6 Conclusion

In this chapter, we showed how to solve name consensus using \(O(1)\)-RMRs per process in the CC and DSM models using \(O(1)\)-RMR leader election as a black box. Name consensus alone (or even leader election for that matter) can be used to construct a “one-shot” Test-And-Set object (i.e., one where each process may apply at most one
operation), as explained in [11]. The key idea behind this implementation is that when a subset of processes calls \texttt{test-and-set}, a leader is elected to decide whose operation will be linearized first. The leader’s operation then succeeds (i.e., returns 0), and all others fail (i.e., return 1).

We use a similar idea to construct implementations of \texttt{CAS} and \texttt{LL/SC}, with two key differences. First, our implementations are long-lived rather than “one-shot”. Still, in any history we can identify groups of (\texttt{CAS} or \texttt{SC}) operations applied concurrently and decide the linearization order for each group by electing a leader. Second, in each such group, operations that are linearized after the leader’s must return the value swapped in by the leader, and must do so only after the leader’s operation has taken effect. In contrast, in the case of Test-And-Set such operations would just return the constant 1, and in a one-shot implementation no additional waiting is needed for linearizability.

The more intricate synchronization required in our implementations of \texttt{CAS} and \texttt{LL/SC} motivates using name consensus instead of leader election. This is because knowing the leader’s ID makes it easier to synchronize using only $O(1)$ RMRs per process in the DSM model. (In the CC model name consensus is very easy to obtain from leader election, as shown earlier in this chapter, and so it offers little advantage.) We show how to perform this type of synchronization in the next chapter, where we introduce another building block that internally uses name consensus. Multiple instances of this building block are used in our \texttt{CAS} and \texttt{LL/SC} implementations.
Chapter 4

Pseudo-Locks

In this chapter, we define a new building block called a *pseudo-lock*, which underlies the implementations presented in later chapters. We then show how to construct pseudo-locks using name consensus, reads and writes at a cost of $O(1)$ RMRs per process in the CC and DSM models.

Informally, a pseudo-lock is similar to a “one-shot” mutex, where at most one process executes the critical section, whereas the others merely wait for the winner to complete the critical section. Formally, a pseudo-lock is an algorithm that consists of two functions: an *entry protocol*, denoted \texttt{Pseudo-Enter()}, and an *exit protocol*, denoted \texttt{Pseudo-Exit()}. The two functions must be accessed according to the following etiquette:

**Condition 4.1.**

(a) Each process calls \texttt{Pseudo-Enter()} and \texttt{Pseudo-Exit()} at most once.

(b) A process can call \texttt{Pseudo-Exit()} only after completing a call to \texttt{Pseudo-Enter()}.

The correctness properties of a pseudo-lock are captured in Specifications 4.2–4.3.

**Specification 4.2** (safety). For any history where Condition 4.1 holds:

(a) If \texttt{Pseudo-Enter()} terminates, it returns a Boolean. If \texttt{Pseudo-Exit()} terminates, it returns \texttt{OK}.

(b) \texttt{Pseudo-Enter()} returns \texttt{true} to at most one process.

(c) If \texttt{Pseudo-Enter()} returns \texttt{false} to some process, then some other process has completed a call to \texttt{Pseudo-Enter()} with response \texttt{true} and subsequently made a call to \texttt{Pseudo-Exit()}.
Chapter 4. Pseudo-Locks

Specification 4.3 (liveness). For any fair history where Condition 4.1 holds:

(a) If at least one call to \texttt{Pseudo-Enter()} is made, then at least one such call terminates.

(b) If some process calls \texttt{Pseudo-Enter()} with response \texttt{true} and then completes a call to \texttt{Pseudo-Exit()}, then all calls to \texttt{Pseudo-Enter()} terminate.

(c) Every call to \texttt{Pseudo-Exit()} terminates.

Definition 4.4. We say that a process acquires the pseudo-lock if it makes a call to \texttt{Pseudo-Enter()} with response \texttt{true}. We say that a process fails to acquire the pseudo-lock if it makes a call to \texttt{Pseudo-Enter()} with response \texttt{false}.

In the remainder of this chapter, we present \(O(1)\)-RMR pseudo-lock implementations for the CC and DSM models.

4.1 Pseudo-locks in the CC Model

A pseudo-lock is straightforward to implement in the CC model given an \(O(1)\)-RMR name consensus algorithm, such as the one described in Chapter 3. One implementation is presented in Figure 4.1. (A very similar algorithm can be devised using leader election instead of name consensus, but we use name consensus nevertheless for consistency with the DSM algorithm.) Let \(A_{PL-CC}\) denote the corresponding concurrent system.

Theorem 4.5. For any history \(H\) of \(A_{PL-CC}\) where Condition 4.1 holds, Specifications 4.2 and 4.3 hold. Furthermore, each call to \texttt{Pseudo-Enter()} or \texttt{Pseudo-Exit()} incurs \(O(1)\) RMRs in the CC model.

Proof. First, note that when Condition 4.1 holds, each process calls \texttt{Pseudo-Enter()} at most once, and so the name consensus algorithm is accessed according to Condition 3.1 (i.e., at most once by each process). Thus, we can appeal to Specifications 3.2 and 3.3 (of name consensus) in our analysis below.

Specification 4.2: Property (a) follows easily from the algorithm. Property (b) holds because \texttt{Pseudo-Enter()} returns \texttt{true} only if the caller won \texttt{NameDecide()} at line 69, which happens for at most one process by Specification 3.2. Property (c) holds because \texttt{Pseudo-Enter()} returns \texttt{false} only after the caller reads \(flag = true\) at line 73,
Declarations

Shared variables: (global)
   flag – Boolean, initially false

Subroutines: (global)
   NameDecide() – $O(1)$-RMR name consensus algorithm (see Chapter 3)

Private variables: (per-process)
   winner – process ID, uninitialized

Function Pseudo-Enter()

Output: Boolean

69 winner := NameDecide()
70 if winner = PID then
71    return true
72 else
73    await flag = true
74    return false
75 end

Function Pseudo-Exit()

// Note: winner is assigned at line 69 of Pseudo-Enter.

76 if winner = PID then
77    write flag := true
78 end
79 return OK

Figure 4.1: Pseudo-lock for the CC model.

which does not happen until the process that won name consensus completes line 77 of Pseudo-Exit(). This process is the one whose Pseudo-Enter() returned true, and which has called Pseudo-Exit() by the time flag = true holds.

Specification 4.3: Let $H$ by any fair history where Condition 4.1 holds. For any call to Pseudo-Enter() in $H$, NameDecide() at line 69 terminates by Specification 3.3. The process that won NameDecide() (which is unique by Specification 3.2) then completes Pseudo-Enter() after $O(1)$ additional steps, which implies property (a). Since the busy-wait loop at line 73 of Pseudo-Enter() terminates after the first write of flag at line 77 of Pseudo-Exit(), property (b) also holds. Finally, property (c) follows directly from the structure of Pseudo-Exit().

RMR complexity: This follows from the structure of Pseudo-Enter() and Pseudo-Exit(), from the RMR complexity of the name consensus algorithm, and from the fact that the busy-wait loop at line 73 terminates after at most two RMRs (one at the first access to flag, and possibly one more once flag is overwritten with true at line 77).
4.2 Pseudo-lock Implementation for the DSM Model

A pseudo-lock is straightforward to implement in the DSM model given an $O(1)$-RMR name consensus algorithm, such as the one described in Chapter 3, multiple instances of a two-process leader election algorithm that can be made local to one process (see Section 3.3), and the functions `signal/wait` from Section 3.4. One implementation is presented in Figure 4.2. Let $A_{PL-DSM}$ denote the corresponding concurrent system.

Declarations

**Shared variables:** (global)

$L2P[1..N][1..N]$ – array of $O(1)$-RMR two-process LE algorithms, where $L2P[p][q]$ is for $p$ and $q$, and local to $p$ (see Section 3.3)

**Subroutines:**

- `NameDecide()` – $O(1)$-RMR name consensus algorithm (see Chapter 3)
- `signal/wait` – subroutines from Section 3.4

**Private variables:** (per-process)

- `winner` – process ID, uninitialized
- `q` – process ID, uninitialized
- `P` – set of process ID, initially $\emptyset$

---

Function `Pseudo-Enter()`

Output: Boolean

```
80 winner := NameDecide()
81 if winner = PID then
82     return true
83 else
84     if $L2P[winner][PID].LeaderElect() = win$ then
85         wait(winner)
86     end
87     return false
88 end
```

Function `Pseudo-Exit()`

```
89 if winner = PID then
90     foreach $q \in P \setminus \{PID\}$ do
91         if $L2P[PID][q].LeaderElect() = lose$ then
92             $P := P \cup \{q\}$
93         end
94     end
95     signal($P$)
96 end
97 return OK
```

---

Figure 4.2: Pseudo-lock for the DSM model.
Theorem 4.6. For any history $H$ of $A_{PL-DSM}$ where Condition 4.1 holds, Specifications 4.2 and 4.3 hold. Furthermore, each call to $Pseudo-Enter()$ or $Pseudo-Exit()$ incurs $O(1)$ RMRs in the DSM model.

Proof. As in the proof of Theorem 4.5, note that the name consensus algorithm is accessed according to Condition 3.1, and so we can appeal to Specifications 3.2 and 3.3. In particular, it follows easily from the algorithm and Specification 3.2 that the two-process LE algorithm instances are accessed (at line 84 and 91) according to Conditions 3.4 and 3.8, and so we can appeal to Specifications 3.5, 3.6, and 3.9.

Specification 4.2: Properties (a) and (b) follow easily from the algorithm and Specification 3.2, as in the proof of Theorem 4.5. Now consider property (c). Suppose that $Pseudo-Enter()$ returns false to some process $p$. Then some process $w \neq p$ won $NameDecide()$ at line 80. If process $p$ lost $L2P[w][p].LeaderElect()$ at line 84, then $w$ won it at line 91 of $Pseudo-Exit()$. Otherwise, $p$ won at line 84 and then completed a call to $wait(w)$ at line 85, and so $w$ must have called $signal(P)$ with $w \in P$ by Lemma 3.10 (c), namely at line 95 of $Pseudo-Exit()$. In either case, $w$ previously completed a call to $Pseudo-Enter()$ with response true (by the success test at line 89), and then called $Pseudo-Exit()$, as wanted.

Specification 4.3: Let $H$ by any fair history where Condition 4.1 holds. As in the proof of Theorem 4.5, property (a) holds since the winner of $NameDecide()$ at line 80 completes its call to $Pseudo-Enter()$. Let $w$ denote this process. Next, consider property (b). Suppose that $w$ makes a call to $Pseudo-Exit()$ after completing its call to $Pseudo-Enter()$. Any process $q \neq w$ that calls $Pseudo-Enter()$ completes line 80 by Specification 3.3, and then begins executing lines 84–87 by the algorithm and Specification 3.2. The call to $L2P[w][q].LeaderElect()$ at line 84 terminates by Specification 3.6. For termination of the call to $wait(w)$ at line 85, it suffices to show that process $w$ calls $signal(P)$ where $P$ is the set of process that call $wait(w)$ at line 85 of $Pseudo-Enter()$ (see Lemma 3.12). (The other hypothesis of Lemma 3.12 is that $w$ does not also call $signal(P')$ with $P \cap P' \neq \emptyset$, which holds since $w$ calls $signal$ at most once.) To that end, note that by the algorithm, $P$ is the set of processes $z$ such that $z \neq w$ (see line 90) and $w$ lost $L2P[w][z]$ at line 91. Consequently, $z \in P$ if and only if $z$ wins $L2P[w][z]$ at line 84 by Specification 3.5, which occurs if and only if $z$ calls $wait$ at line 85 (since $H$ is fair). The argument of the latter call is $w$ by the algorithm and Specification 3.2. Thus, $q$’s call to $wait$ at line 85 terminates by Lemma 3.12. Finally, property (c) follows from
the structure of \texttt{Pseudo-Exit()}, Specification 3.6, and the termination of \texttt{signal} (see Lemma 3.10 (b)).

**RMR complexity:** This follows from the structure of \texttt{Pseudo-Enter()} and \texttt{Pseudo-Exit()}, the RMR complexity of \texttt{NameDecide()} (given that Condition 3.1 holds), the RMR complexity of the two-process LE algorithms (given that Condition 3.4 holds), the locality of these algorithms (Specification 3.9), and the RMR complexity of \texttt{signal}/\texttt{wait} (Lemma 3.10 (a)).

4.3 Conclusion

In this chapter we introduced pseudo-locks. As hinted at the end of Chapter 3, pseudo-locks will be used later on in our implementations of CAS and LL/SC to linearize groups of concurrent \texttt{CAS} or \texttt{SC} operations. To that end, the response of \texttt{Pseudo-Enter()} can be used to elect a leader (see Specification 4.2 (b)) and hence determine whose operation will be linearized next. Pseudo-locks also address two other issues mentioned at the end of Chapter 3. First, any process $p$ other than the leader must wait for the leader’s operation to take effect before its operation returns; this is addressed by Specification 4.2 (c). Second, $p$ must discover the value swapped in by the leader’s operation; this is straightforward given Specification 4.2 (b)–(c) because the leader can simply record this value in a shared variable between its calls to \texttt{Pseudo-Enter()} and \texttt{Pseudo-Exit()}, and $p$ may then read this value after completing its call to \texttt{Pseudo-Enter()}.

Before we present our implementations of CAS and LL/SC, we introduce in the next chapter another building block. This building block, which exploits pseudo-locks, serves as a common base for all of our implementations later on.
Chapter 5

Block Manager

Our implementations of CAS and LL/SC manipulate data structures we call blocks. These are inspired by “cells” in Herlihy’s universal wait-free construction [14], and are similar to the “blocks” used in the wait-free implementation of CAS by Luchangco, Moir and Shavit [21]. Blocks record the state of the target object, where the current state is stored in a specially designated current block. A process effects a state change using the “pointer-swinging” technique: it allocates a new block representing the new state, and designates that block as current. We say that a block is fresh if all its fields (i.e., the objects contained in it) are in their initial states. At initialization, a fresh block called the initial block is current.

Fresh blocks are allocated using a block allocator, denoted in our pseudo-code by the subroutine AllocBlock(). For now, we assume that this function returns a unique fresh block different from the initial block. (For a discussion of memory recycling, see Chapter 11.) That is, fresh blocks are drawn from an unbounded set, which can be maintained by each process using a private linked list and accessed using only local computation. Formally, we assume that calls to AllocBlock() incur $O(1)$ RMRs and satisfy the following properties:

**Specification 5.1** (safety). For any history:

(a) if a call to AllocBlock() returns response $x$ then $x$ is a block address never before returned by AllocBlock() and different from the initial block (of the block manager); and

(b) a call to AllocBlock() does not access any block.

**Specification 5.2** (liveness). For any fair history, each call to AllocBlock() terminates.
The current block is tracked using a typed shared object we call the block manager. The block manager type, \( \tau_{BM} = (S, s_{init}, O, R, \delta) \), is formally defined as follows. Each state in \( S \) is a tuple \((C, S)\) where \( C \) denotes the address of the current block, and \( S \) is a set of pairs of the form \((x, p)\), where \( x \) is a block address and \( p \) is a process ID. The initial state is \((b_0, \emptyset)\) where \( b_0 \) is the address of the initial block. The following operation types are defined: getCurBlock() and chngCurBlock\((x, y)\). The set of responses consists of the set of block addresses and the set of process IDs. The transition relation is defined by the (atomic execution of) the pseudo-code shown in Figure 5.1. Operation getCurBlock() returns the address of the current block. Operation chngCurBlock\((x, y)\) makes \( y \) the current block, unless some process already called chngCurBlock\((x, ...)\), in which case it has no side-effect. (Here and subsequently we use “...” as a “wildcard” symbol.) We say that a chngCurBlock operation is successful in the first case and failed otherwise. A successful chngCurBlock\((x, y)\) returns the caller’s ID. A failed chngCurBlock\((x, y)\) necessarily follows a successful chngCurBlock\((x, ...)\), and returns the ID of the process that executed the latter operation.

Note that, as specified, a successful chngCurBlock\((x, y)\) does not necessarily change the address of the current block from \( x \) to \( y \); it merely ensures that \( y \) is current. However, later on when we use the block manager to implement CAS and LL/SC, we will call chngCurBlock\((x, y)\) in such a way that if it succeeds then it does change the current block from \( x \) to \( y \); see Lemma 6.6.

```plaintext
Function getCurBlock()
Output: address of current block
98 return C

Function chngCurBlock\((x, y)\)
Input: \( x, y \) – blocks
Output: ID of process whose call to chngCurBlock\((x, ...)\) succeeded
99 if \((x, p) \in S\) for some \( p \) then
100 return \( p \)
101 else
102 \( C := y \)
103 \( S := S \cup \{(x,\text{PID})\} \)
104 return \( \text{PID} \)
105 end
```

Figure 5.1: Definition of block manager operation types. (The current state is denoted by \((C, S)\).)

Finally, we require that any implementation of the block manager satisfy the following:
Specification 5.3 (safety). Any history of the implementation is linearizable with respect to type $\tau_{BM}$.

Specification 5.4 (liveness). In any fair history of the implementation, each call to getCurBlock or chngCurBlock terminates.

5.1 Linearizable $O(1)$-RMR Implementation

We now present a simple implementation of the block manager that uses only reads and writes, and has $O(1)$ RMR complexity in the CC and DSM models. We refer to this implementation as $I_{BM} = (\tau_{BM}, P, B, H)$. The implementation records the address of the current block in a shared register variable $D$. It also relies on an $O(1)$-RMR pseudo-lock, as described in Chapter 4. We use multiple “instances” of the pseudo-lock (one per block), each with its own copies of the underlying shared variables. The pseudo-lock is needed for synchronization inside chngCurBlock, namely when multiple processes apply chngCurBlock$(x, ...)$ operations for some block $x$. In that case, processes use the pseudo-lock in block $x$ to decide whose operation will succeed, and to discover the ID of the process whose operation succeeded. The access procedures for getCurBlock and chngCurBlock are presented in Figure 5.2.

We will now show that the implementation $I_{BM}$ is linearizable, satisfies the termination property, and has $O(1)$ RMR complexity (see Chapter 2).

Lemma 5.5. For any history $H$ of $I_{BM}$ and for any block $x$ accessed by any process in $H$:

(a) The pseudo-lock in block $x$ is accessed according to Condition 4.1.

(b) A read of $x \triangleright winner$ at line 114 of chngCurBlock returns the ID of the unique process that acquired the pseudo-lock in block $x$ and then completed line 109 during the same chngCurBlock operation execution.

Proof.

Part (a): Since a process calls Pseudo-Exit only at line 111, and only after calling Pseudo-Enter at line 108, it suffices to show that $p$ accesses the pseudo-lock in block $x$ at most once. Suppose, for contradiction, that $H$ is the shortest history at the end of which $p$ is about to call $x \triangleright$ Pseudo-Enter() for the second time. The first time $p$ calls $x \triangleright$ Pseudo-Enter() in $H$, this happens at line 108, and before $p$ reaches line 114 in
Declarations

**Shared variables**: (global)

- $D$ – register, stores a block address, initially points to the initial block defined for $\tau_{BM}$

**Shared variables**: (per-block)

- $winner$ – register, stores a process ID or $\perp$, initially $\perp$

**Subroutines**: (one instance per-block)

- Pseudo-Enter() / Pseudo-Exit() – $O(1)$-RMR pseudo-lock from Chapter 4

Function getCurBlock()

```plaintext
return read($D$)
```

Function chngCurBlock($x$, $y$)

```plaintext
if read($x \bowtie winner$) = $\perp$ then
  if $x \bowtie$ Pseudo-Enter() = true then
    write $D := y$
    write $x \bowtie winner := PID$
    $x \bowtie$ Pseudo-Exit()
  end
end
return read($x \bowtie winner$)
```

Figure 5.2: Block manager implementation.
the corresponding call to \texttt{chngCurBlock}, some process assigns a value different from $\bot$ to $x \triangleright winner$ at line 110. This is because either $p$ does so, or by Specification 4.2 (c) (and minimality of $|H|$) some other process does so and then calls $x \triangleright \text{Pseudo-Exit()}$ at line 111 before $p$ completes its call to $x \triangleright \text{Pseudo-Enter()}$. Thus, if $p$ subsequently accesses block $x$ inside \texttt{chngCurBlock}, then $x \triangleright winner \neq \bot$ holds at line 107, and so $p$ does not access the pseudo-lock in block $x$ at all. This contradicts the hypothesis that $p$ is about to call $x \triangleright \text{Pseudo-Enter()}$ for the second time at the end of $H$.

**Part (b):** Consider the value a process reads from $x \triangleright winner$ at line 114 of \texttt{chngCurBlock}. Since we showed above that the pseudo-lock in block $x$ is accessed according to Condition 4.1, it follows from the algorithm (Figure 5.2) that at most one process writes $x \triangleright winner$, namely the winner of the pseudo-lock in block $x$. (We appeal to Specification 5.1 (b) implicitly here and in many other proofs.) We also argued above that by the time a process reaches line 114, some process has executed line 110, hence line 109. These two observations imply the lemma. \hfill $\square$

To prove linearizability, we define for each history $H$ of $I_{BM}$ a candidate linearization $\bar{H}$ as follows. First, for each operation execution on the target object in $H$, we assign a numerical “timestamp”.

**Definition 5.6.** The timestamp $s$ for an arbitrary operation execution $Op$ in $H$, say by process $p$, and its completion (where applicable), are defined as follows:

**Operation type \texttt{getCurBlock}():**

(a) If $Op$ is complete in $H$ and $p$ reads $D$ at line 106 during $Op$ in step $i$ of $H$, then $s = i$.

(b) Otherwise $s$ is undefined, and $Op$ does not appear in $\bar{H}$.

**Operation type \texttt{chngCurBlock}(x, y):**

(c) If $p$ writes $D$ at line 109 of \texttt{chngCurBlock} during $Op$ in step $i$ of $H$, then $s = i$.

(The completion of $Op$, if $Op$ is pending in $H$, returns $p$’s ID.)

(d) Else if $Op$ is complete and $p$ does not write $D$ at line 109 during $Op$, and reads $x \triangleright winner$ at line 114 for some block $x$ during $Op$ in step $i$ of $H$, then $s = i$.

(e) Otherwise $s$ is undefined, and $Op$ does not appear in $\bar{H}$.
To construct our candidate linearization $\tilde{H}$ of $H$, we arrange operation executions for which timestamps are defined, in increasing order of timestamp. (The uniqueness of these timestamps follows easily from Definition 5.6.) Operation executions that are pending in $H$, and whose timestamps are defined, are completed as explained above.

**Lemma 5.7.** $\tilde{H}$ satisfies properties (a) and (b) of linearizability (sequential completion and order preservation).

**Proof.** Property (a) follows from our construction of $\tilde{H}$ and Definition 5.6. For property (b), note that by Definition 5.6, if the timestamp of an operation execution $Op$ by $p$ in $H$ is $i$, then $p$ executes step $i$ during $Op$ in $H$. Thus, if $Op$ and $Op'$ are operation executions in $H$ whose counterparts appear in that order in $\tilde{H}$, then $Op$ has a smaller timestamp, and so either $Op$ and $Op'$ are concurrent in $H$, or $Op$ precedes $Op'$ in $H$. $\square$

It remains to prove property (c) (conformity to type $\tau_{BM}$). To that end, we first define some useful notation. Let $Op_i$, $s_i$, and $p_i$ denote the $i$'th operation execution in $\tilde{H}$ (counting from 1), its timestamp, and the calling process. If $Op_i$ is a chngCurBlock operation execution, then we will refer to the arguments of $Op_i$ as $x_i$ and $y_i$. Finally, let $\nu_i = (C_i, S_i)$ denote the state of the block manager after applying the first $i$ operation executions in $\tilde{H}$ on a correctly implemented block manager initialized to $b_0$ (the initial block).

**Lemma 5.8.** Implementation $I_{BM}$ satisfies property (c) of linearizability (conformity to type $\tau_{BM}$).

**Proof.** Let $H$ be any history of $I_{BM}$. Since conformity to a type is a safety property it suffices to consider finite $\tilde{H}$. Let $k = |\tilde{H}|$. Define $s_0 = 0$ and $s_{k+1} = \infty$. We will prove that for any $i \in \mathbb{N}$, $0 \leq i \leq k$:

(a) For any integer $t \in [s_i, s_{i+1})$, $D = C_i$ holds in state $H[t]$.

(b) If $i > 0$, then the response of $Op_i$ is the correct response for an operation execution of that type applied in state $\nu_{i-1}$.

Part (b) implies the lemma, but we require both parts for induction. Now let $S(i)$ denote parts (a)–(b) for a particular value of $i$. Note that in $H$, the value of $D$ is changed only by an execution of line 109, which is an atomic step that defines the timestamp of an operation execution (on the target object) in $\tilde{H}$. Therefore, the value of $D$ does not
change between atomic steps $s_i$ and $s_{i+1}$ in $H$. This, in turn, implies that to prove part (a) of $S(i)$, it suffices to prove that $D = C_i$ in state $H[s_i]$ — and that is all we do in the inductive step that follows.

For $S(0)$, (a) follows from our earlier definition of $s_0$, and the initialization of $D$ to $b_0$ (the initial block). Part (b) holds trivially for $S(0)$. Now for any $i$, $0 < i \leq k$, suppose that $S(i-1)$ holds, and consider $S(i)$. We proceed by cases on how the timestamp $s_i$ was obtained.

**Case A:** $O_{p_i}$ is a getCurBlock operation execution and process $p_i$ reads $D$ at line 106 in step $s_i$ of $H$. In this case, $\nu_i = \nu_{i-1}$.

$S(i)$ part (a) follows from $S(i-1)$ part (a) because $\nu_i = \nu_{i-1}$ and step $s_i$ in $H$ does not write $D$.

For $S(i)$ part (b), note that by the algorithm, $O_{p_i}$ returns in $\bar{H}$ the value $p_i$ that read from $D$ in step $s_i$. By $S(i-1)$ part (a), this value equals $C_{i-1}$, which is the correct response for $O_{p_i}$.

**Case B:** $O_{p_i}$ is a chngCurBlock($x_i$, $y_i$) operation execution and process $p_i$ wrote $D$ at line 109 in step $s_i$ of $H$.

Since $p_i$ acquires the pseudo-lock in block $x_i$ during the counterpart of $O_{p_i}$ in $H$, it follows from Lemma 5.5 (a) and Specification 4.2 (b) that no other process does so in $H$. Furthermore, by Lemma 5.5 (a) and Specification 4.2 (c), no process completes a call to $x_i \triangleright \text{Pseudo-Enter()}$ at line 108 before step $s_i$ in $H$. Consequently, it follows from Definition 5.6 and our construction of $\bar{H}$ that $O_{p_i}$ is the first chngCurBlock($x_i$, ...) operation execution in $\bar{H}$. Thus, $O_{p_i}$ succeeds, and so $\nu_i = (C_i, S_i)$ where $C_i = y_i$ and $S_i = S_{i-1} \cup \{(x_i, p_i)\}$.

$S(i)$ part (a) follows by the action of step $s_i$ in $H$, where $p_i$ writes $y_i = C_i$ to $D$.

For $S(i)$ part (b), we must show that $O_{p_i}$ returns $p_i$’s ID since it succeeds. But this follows from line 114 and Lemma 5.5 (b).

**Case C:** $O_{p_i}$ is a complete chngCurBlock($x_i$, $y_i$) operation execution and process $p_i$ reads $x_i \triangleright \text{winner}$ at line 114 in step $s_i$ of $H$, but does not write $D$ at line 109 during the counterpart of $O_{p_i}$ in $H$.

Since $p_i$ does not acquire the pseudo-lock in block $x_i$ and completes line 108 during the counterpart of $O_{p_i}$ in $H$, we will show that some chngCurBlock($x_i$, ...), precedes
If \( p_i \) reads \( x \triangleright \text{winner} \neq \perp \) at line 107, then some process wrote \( x \triangleright \text{winner} \) at line 110 before step \( s_i \) in \( H \). This happens during the counterpart of some \( \text{chngCurBlock}(x_i,...) \) operation execution \( Op_j \) in \( H \) that falls under Case B above, and precedes \( Op_i \) by the order of lines 109–110, by Definition 5.6, and by our construction of \( H \). On the other hand, if \( p_i \) executes line 108 and fails to acquire the pseudo-lock, then it follows from Lemma 5.5 (a) and Specification 4.2 (c) that some other process \( q \) does acquire the same pseudo-lock, and then makes a call to \( x_i \triangleright \text{Pseudo-Exit}() \) before step \( s_i \) in \( H \). This happens during the counterpart of some \( \text{chngCurBlock}(x_i,...) \) operation execution \( Op_j \) that falls under Case B above, and precedes \( Op_i \) by Definition 5.6 and by our construction of \( H \). Thus, \( Op_i \) is a failed \( \text{chngCurBlock} \), and so \( \nu_i = \nu_{i-1} \).

\( S(i) \) part (a) follows from the fact that \( \nu_{i-1} = \nu_i \) and the action of step \( s_i \) in \( H \), which does not overwrite \( D \).

For \( S(i) \) part (b), we must show that \( Op_i \) returns the ID of the process that applies a successful \( \text{chngCurBlock}(x_i,...) \) in \( H \). Recall that \( Op_i \) returns the ID \( p_i \) reads from \( x_i \triangleright \text{winner} \) at line 114, which by Lemma 5.5 (b) is the ID of the unique process \( q \) that acquired the pseudo-lock in block \( x_i \) and then executed line 109. The corresponding \( \text{chngCurBlock}(x_i,...) \) operation execution by \( q \) appears in \( \tilde{H} \) by Definition 5.6 and our construction of \( \tilde{H} \), and is successful by our analysis of Case B, as wanted.

\( \square \)

**Theorem 5.9.** The implementation \( I_{BM} \) of the block manager satisfies Specifications 5.3 and 5.4. Furthermore, for any history \( H \) of \( I_{BM} \), each call to \( \text{getCurBlock} \) or \( \text{chngCurBlock} \) incurs \( O(1) \) RMRs in the CC and DSM models.

**Proof.** Let \( H \) be any history of \( I_{BM} \).

**Specification 5.3:** Linearizability of \( H \) follows from Lemma 5.7 and Lemma 5.8 (b).

**Specification 5.4:** If \( H \) is fair, each call to \( \text{getCurBlock} \) terminates by the structure of the access procedure. Similarly, each call to \( \text{chngCurBlock} \) terminates provided that the pseudo-lock functions \( \text{Pseudo-Enter} \) and \( \text{Pseudo-Exit} \) terminate. For termination of the latter functions, first recall that by Lemma 5.5 (a), Condition 4.1 holds with respect to any pseudo-lock accessed in \( H \). Since Condition 4.1 holds, and since any process that
acquires a pseudo-lock in any block $x$ eventually calls $x\triangleright Pseudo$-Exit(), the pseudo-lock functions terminate by Specification 4.3.

**RMR complexity:** The RMR complexity of $I_{BM}$ follows from the structure of the access procedures and the RMR complexity of the pseudo-lock functions (where Condition 4.1 holds by Lemma 5.5 (a)).

### 5.2 Conclusion

In this chapter we introduced the block manager and block allocator. The main purpose of the block manager is to encapsulate the somewhat complex synchronization algorithms needed by our implementations of CAS and LL/SC, which we present starting in Chapter 6. When these implementations are expressed with respect to a set of atomic base objects that includes a block manager, their access procedures are wait-free, which simplifies analysis greatly.
Chapter 6

Extended Compare-and-Swap

Our goal in this chapter is to provide $O(1)$-RMR implementations, using reads and writes only, of two well-known synchronization primitives: compare-and-swap (CAS) and load-linked/store-conditional (LL/SC). We first give precise definitions of these primitives as shared object types, and then describe our implementation methodology. Our definitions at this stage are simplified in the sense that the type for each primitive does not support a Write operation type; we defer discussion of writable objects to Chapter 8.

We model the CAS primitive as a shared object type $\tau_{\text{CAS}}$. The set of states of the type is the set of values from a domain $U$, and the initial state $s_{\text{init}}$ can be any element of $U$. The following operation types are defined: $\text{CAS}(\text{cmp}, \text{new})$ and $\text{Read}()$. The set of responses is also $U$. The transition relation is defined by the (atomic execution of) the pseudo-code shown in Figure 6.1. $\text{CAS}(\text{cmp}, \text{new})$ returns the prior state, which is a value from $U$, and also acts on the state according to one of two execution paths: A successful CAS operation occurs when the prior state is $\text{cmp}$, in which case it changes the state to $\text{new}$. A failed CAS operation occurs when the prior state is different from $\text{cmp}$, in which case the state does not change. $\text{Read}()$ simply returns the previous state and does not change the state.

Load-linked/store-conditional (LL/SC) is another popular synchronization primitive, and is similar in spirit to CAS. We model LL/SC as a shared object type $\tau_{\text{LL/SC}}$ whose state consists of a value $V$ from a domain $U$ and a Boolean array $\text{Linked}[1..N]$. In the initial state $s_{\text{init}}$, $V$ can be any element of $U$, and each element of $\text{Linked}[1..N]$ is false. The following operation types are defined: $\text{LL}()$, $\text{SC}(\text{new})$ and $\text{Read}()$. The set of responses consists of the elements of $U$ (for LL and Read) and the Boolean constants $\{\text{true}, \text{false}\}$ (for SC). The transition relation is defined by (the atomic execution of) the
Figure 6.1: Definition of operation types for type $\tau_{CAS}$. (The current state is denoted by $V$.)

Both CAS and LL/SC are important and commonly implemented (in hardware) primitives. (Popular architectures such as x86, Itanium, Sparc, MIPS, IBM Power, and DEC Alpha support a variant of either CAS or LL/SC.) They are typically used on a
shared variable by calling \texttt{Read} or \texttt{LL} first to retrieve the value, say $v$, and then calling \texttt{CAS} or \texttt{SC} to try changing the value from $v$ to some $v'$. In some applications, \texttt{LL/SC} is preferred over \texttt{CAS} because it does not suffer from the so-called A-B-A problem. That is, \texttt{SC} can “detect” when the value of an object has changed (say from A to B) since the last call to \texttt{LL}, and then changed back (from B to A), whereas \texttt{CAS} only “looks” at the latest value and succeeds or fails accordingly.

Although \texttt{CAS} and \texttt{LL/SC} are similar in spirit, due to the subtle difference between them, it is difficult to simulate one from the other in a manner that preserves the RMR-related correctness properties under consideration in this thesis. (Constant-time wait-free simulations of \texttt{LL/SC} from \texttt{CAS} and vice-versa are known, e.g. [23, 16], but are insufficient for our purposes because they either use registers of unbounded size or they do not provide special “locality properties” defined later on in Chapter 7.) Rather than showing how to implement each type separately, we show how to implement a stronger object type called \textit{extended compare-and-swap} (ECAS), from which \texttt{CAS} and \texttt{LL/SC} can be derived very easily. This new type, denoted $\tau_{\text{ECAS}}$, provides an operation type \texttt{ECAS} that behaves either like \texttt{CAS} or like \texttt{SC} depending on the value of a Boolean parameter (called \textit{isSC}). Like \texttt{CAS} and \texttt{SC}, \texttt{ECAS} either succeeds or fails, which is indicated in its response.

The state space and initial value of $\tau_{\text{ECAS}}$ are defined as for $\tau_{\text{LL/SC}}$ (i.e., the combination of a value $V$ and a Boolean array $\text{Linked}[1..N]$). There are three operation types: \texttt{Read}, \texttt{LL} and \texttt{ECAS}(\textit{isSC}, \textit{cmp}, \textit{new}). The set of responses consists of the elements of $U$ (for \texttt{LL} and \texttt{Read}), as well as the set of ordered pairs of the form $(v, b)$ where $v \in U$ and $b \in \{\text{true, false}\}$ (for \texttt{ECAS}). The transition relation is defined by (the atomic execution of) the pseudo-code shown in Figure 6.2 (for \texttt{LL} and \texttt{Read}) and in Figure 6.3 (for \texttt{ECAS}). \texttt{ECAS} is the new operation type that generalizes \texttt{SC} and \texttt{CAS}, and corresponds to the atomic execution of the pseudo-code shown in Figure 6.3. \texttt{ECAS} can simulate either \texttt{SC} or \texttt{CAS} depending on the value of the parameter \textit{isSC}, as we explain below. It returns a pair $(v, b)$ consisting of the prior value $v$ and a Boolean success indicator $b$, which we also explain below.

The conditional statement in Figure 6.3 has two cases. In the first case, \texttt{ECAS} behaves either like a failed \texttt{SC}(\textit{new}) operation (with $\text{Linked}[\text{PID}] = \text{false}$) or like a failed \texttt{CAS}(\textit{cmp}, \textit{new}) operation (with $\text{cmp} \neq V$), leaving $V$ and $\text{Linked}[1..N]$ unchanged. In the second case, \texttt{ECAS} behaves like a successful \texttt{CAS}(\textit{cmp}, \textit{new}) or \texttt{SC}(\textit{new}) operation, assigning $V = \text{new}$ and also resetting $\text{Linked}[1..N]$. In both cases, the response is a
Function ECAS(isSC, cmp, new)

Input: isSC – Boolean, cmp – comparison value, new – value to be swapped in

Output: pair (prior value, Boolean success indicator)

old := V

if (isSC = true ∧ Linked[PID] = false) ∨ (isSC = false ∧ cmp ≠ old) then

    return (old, false) // Operation failed.

else

    V := new

    foreach i ∈ 1..N do Linked[i] := false

    return (old, true) // Operation successful.

end

Figure 6.3: Definition of ECAS operation type for type $\tau_{ECAS}$. (The current state is denoted by $V$ and $Linked[1..N]$.)

A tuple containing the prior value and a success indicator; we say that an ECAS operation is successful in the second case, and failed otherwise. (Note: At line 138 of Figure 6.3, it is not always necessary to reset all elements of $Linked[1..N]$, but we do so anyway for simplicity. RMR complexity is not relevant in this context since we are defining the transition relation for a shared object type rather than implementing that type.)

Implementing CAS and LL/SC (individually) is straightforward given a single ECAS base object. These implementations, which have the same RMR complexity (asymptotically) as the underlying ECAS object, are presented in Figure 6.4. In general, our implementations will satisfy the following properties:

**Specification 6.1** (safety). Any history of the implementation is linearizable with respect to the primitive’s type.

**Specification 6.2** (liveness). In any fair history of the implementation, each call to an access procedure terminates.

The correctness properties of the implementations presented in Figure 6.4 are captured by the following theorem:

**Theorem 6.3.** Let $\tau$ be one of $\tau_{CAS}$ or $\tau_{LL/SC}$. The implementation $I$ of $\tau$ presented in Figure 6.4 satisfies the following correctness properties:

(a) **Specification 6.1** (linearizability with respect to type $\tau$).
Declarations

**Shared variables:** (global)
- $B$ – ECAS object

**Private variables:** (per-process)
- $val$ – value from domain $U$, uninitialized
- $succ$ – Boolean, uninitialized

Function $\text{Read}()$

**Output:** current value

return $B.\text{Read}()$

Function $\text{CAS}(cmp, new)$

**Input:**
- $cmp$ – comparison value
- $new$ – value to be swapped

**Output:** prior value

$\text{(val, succ)} :=$

$B.\text{ECAS}(false, cmp, new)$

return $val$

Function $\text{LL}()$

**Output:** current value

return $B.\text{LL}()$

Function $\text{SC}(new)$

**Input:** $new$ – value to be stored

**Output:** Boolean success indicator

$\text{(val, succ)} :=$

$B.\text{ECAS}(true, new, new)$

return $succ$

---

Figure 6.4: Implementations of CAS and LL/SC from ECAS.
(b) Specification 6.2 (termination).

(c) Operation executions on the target object have the same worst-case RMR complexity (asymptotically) as atomic steps on the base object $B$.

Proof. Specification 6.1 (linearizability) follows easily if we consider that an operation execution on the target object takes effect at the same point as the corresponding atomic step on the base object $B$. Specification 6.2 (termination) and RMR complexity follow directly from the structure of the access procedures, each of which applies only a single atomic step, namely on the base object $B$. □

In the remainder of this chapter, we present a $O(1)$-RMR implementation of ECAS using reads and writes only. The implementation is linearizable but only in certain histories; this restriction simplifies the underlying algorithms, but at the same time allows the ECAS object to be used for implementing CAS and LL/SC (individually), as shown in Figure 6.4. The particular histories under consideration are those satisfying the following condition:

**Condition 6.4.** Either no process invokes LL, or no process invokes ECAS$(isSC, cmp, new)$ with $isSC = false$.

### 6.1 Linearizable $O(1)$-RMR Implementation of ECAS

We now present an implementation $I_E = (\tau_{ECAS}, P, B, H)$ of ECAS that uses a block manager, as described in Chapter 5, as its principal building block. The implementation uses blocks to record the state of the target object. To that end, each block contains fields called $V$ and $Linked[1..N]$, which correspond to the two components of the target object’s state. Whenever a successful ECAS operation execution occurs, the caller allocates a new block and makes that block current by calling $\text{chgCurBlock}$. The latter operation execution performs much of the synchronization needed to handle concurrent ECAS operation executions by deciding which operation execution will succeed and which will fail. Two other fields are present in each block. First, an array $NextVal[1..N]$ is used (in some cases) by a successful ECAS operation execution to communicate the new value of the target object to a failed ECAS operation execution that must return that value. Second, a register $writer$ is used to determine the ID of the process that allocated and made current a particular block.
The access procedures for the operation types Read, LL, and ECAS are presented in Figures 6.5–6.7. Lines containing shaded statements can be ignored safely for now; these statements come into play in Chapter 7 when we discuss locally-accessible implementations. For completeness, we provide in Figure 6.7 stub implementations of the subroutines called in shaded statements.

The access procedures for Read, LL, and ECAS are designed around the following invariant: the current state of the target object is stored in the variables V and Linked[1..N] in the current block. The access procedure for Read simply retrieves the current block, say x, and returns the value read from x ⊲ V. LL is similar, but also ensures that x ⊲ Linked[PID] = true holds. The access procedure for ECAS begins at lines 160–162 in the same way as Read and LL. Then, at lines 163–166, certain special cases are tested, which allow ECAS to return without trying to change the current block; these are the cases when the operation execution fails, or succeeds without changing the state of the target object. Lines 167–169 do nothing in this version of the implementation; they come into play only in Section 7.2.

If the execution of ECAS reaches line 171, then the calling process has a chance to execute a successful ECAS, which will change the state of the target object. Since there may be several such operation executions that access the same block x, we allow one of them to succeed, and force the rest to fail. (We can always do so thanks to lines 163–166, which for this reason are not merely an optimization.) The successful operation execution is the one whose chngCurBlock(x, ...) succeeds at line 175, after the caller allocates and initializes a new block at lines 171–173. This operation execution returns the value of its cmp argument and a success indicator of true. (Note that a successful chngCurBlock(x, y) ensures that y.Linked[1..N] = false because this array in block y has not been accessed since initialization to false. Thus, such a successful chngCurBlock is the counterpart of line 138 in Figure 6.3.)

Prior to calling chngCurBlock(x, ...) at line 175, each process records its argument new in its entry of the array x ⊲ NextVal[1..N] at line 174. By doing so, each process ensures that if its subsequent chngCurBlock(x, ...) succeeds then the new value of the target object is already recorded in block x, and so it can be accessed easily at line 177 by processes whose chngCurBlock(x, ...) fails. The ECAS operation executions of processes in the latter category fail and return this new value (as well as a success indicator of false) without any additional waiting.

The high-level approach underlying our ECAS implementation cannot be used to
### Declarations for ECAS implementation.

#### Shared variables: (global)
- \(M\) – \(O(1)\)-RMR block manager (see Chapter 5)

#### Subroutines: (global)
- `AllocBlock()` – \(O(1)\)-RMR block allocator

#### Shared variables: (per-block)
- \(V\) – register, stores value from domain \(U\), initialized to the initial value of type \(\tau_{ECAS}\)
- `Linked[1..N]` – array of Boolean, initially all `false`, element \(i\) private to process \(i\)
- `NextVal[1..N]` – array of registers, same type as \(V\), uninitialized
- `writer` – register, stores process ID or \(\bot\), initially \(\bot\)

#### Private variables: (per-process)
- `old` – value from domain \(U\), uninitialized
- `ret` – Boolean, uninitialized
- `d, d'` – block addresses, uninitialized
- `winner` – process ID, uninitialized

---

**Function** `Read()`

```plaintext
147 d := M.getCurBlock()
148 HelperBegin(d)
149 old := read(d \triangleright V)
150 HelperEnd(d)
151 return old
```

**Function** `LL()`

```plaintext
152 d := M.getCurBlock()
153 HelperBegin(d)
154 old := read(d \triangleright V)
155 if read(d \triangleright Linked[PID]) = false
156    then
157        write d \triangleright Linked[PID] := true
158    end
159 HelperEnd(d)
160 return old
```

Figure 6.5: Implementation \(I_E\) of ECAS – part 1.
Function ECAS(isSC, cmp, new)

d := M.getCurBlock()
HelperBegin(d)
old := read(d ⊲ V)
if (isSC = true ∧ read(d ⊲ Linked[PID]) = false) ∨ (isSC = false ∧ cmp ≠ old) then
    // Operation execution failed.
    ret := false
else if isSC = false ∧ cmp = new then
    // Operation execution successful, does not change the state.
    ret := true
else if HelperCC(d, new) = true then
    // Operation execution successful, changes the state
    // without changing the current block.
    ret := true
else
    // Try to execute successful operation execution that changes the state.
    d′ := AllocBlock()
    write d′ ⊲ writer := PID
    write d′ ⊲ V := new
    write d ⊲ NextVal[PID] := new
    winner := M.chngCurBlock(d, d′)
    if winner ≠ PID then
        // Operation execution failed.
        old := read(d ⊲ NextVal[winner])
        ret := false
    else
        // Operation execution successful, changes the state.
        ret := true
    end
HelperEnd(d)
return (old, ret)

Figure 6.6: Implementation $I_E$ of ECAS – part 2.
obtain $O(1)$-RMR implementations of Fetch-And-Add or Fetch-And-Store – primitives that are very powerful for solving mutual exclusion efficiently with respect to RMRs. The approach breaks down because it assumes that when many processes apply operations on the implemented object concurrently, they can be linearized in some order where at most one of them changes the state of the implemented object. This requires fairly weak symmetry-breaking power, particularly no more than is provided by each process calling \texttt{chngCurBlock} once (i.e., a single “round” of leader election). In contrast, if $N$ processes apply Fetch-And-Add or Fetch-And-Store operations concurrently then in the worst case all $N$ operations change the state of the implemented object, and one of $N!$ possible orderings among these operations must be decided. This would require each operation to repeatedly find the current block by calling \texttt{getCurBlock()} and try to change it by calling \texttt{chngCurBlock}, until such a call to \texttt{chngCurBlock} succeeds. Since a \texttt{chngCurBlock} is not guaranteed to succeed, this means that starvation would be possible.

### 6.1.1 Analysis

We begin our analysis by proving linearizability. Then, we consider the RMR complexity and termination properties.

**Lemma 6.5.** For any history $H$ of implementation $I_E$, and for any block $x$, if $x$ is current at some point in $H$, and subsequently an atomic step involving a successful $M.chngCurBlock(...,...)$ occurs, then $x$ is not current at any point after that step in $H$.

**Proof.** Suppose otherwise, and consider the second time $x$ becomes current. (If $x$ is the initial block, we count initialization as the first time.) This happens by the action of a successful $M.chngCurBlock(...,x)$ at line 175, and $x$ is the value returned by \texttt{AllocBlock()} at line 171. Consequently, by Specification 5.1, $x$ is not the initial block, and so the first time $x$ became current was also by a successful $M.chngCurBlock(...,x)$, preceded by
another call to \texttt{AllocBlock()} that returned \(x\). Thus, two calls to \texttt{AllocBlock()} return \(x\) in \(H\), which contradicts Specification 5.1.

\textbf{Lemma 6.6.} For any history \(H\) of implementation \(I_E\), for any blocks \(x\) and \(y\), and for any positive \(k \in \mathbb{N}\), if a successful \(M.\texttt{chngCurBlock}(x, y)\) occurs in step \(k\) of \(H\), then \(x\) is current in state \(H[k-1]\).

\textit{Proof.} Let \(b_i\) denote the \(i\)'th block that becomes current (the initial block being \(b_1\)), and note that if \(i > 1\) then \(b_{i-1}\) is current just before \(b_i\) becomes current. Suppose for contradiction that the lemma is false, and consider the smallest \(i > 1\) such that block \(b_i\) becomes current in step \(k\) of \(H\) by way of a successful \(M.\texttt{chngCurBlock}(b, b_i)\) where \(b \neq b_{i-1}\). Since \(b\) is the value returned by \(M.\texttt{getCurBlock()}\) before this \(\texttt{chngCurBlock}\), it follows that \(b = b_j\) for some \(j < i\). Now consider the subhistory \(H'\) of \(H\) after this \(\texttt{getCurBlock}\) and before step \(k\). Since \(b \neq b_{i-1}\), it follows by the minimality of \(i\) that a successful \(M.\texttt{chngCurBlock}(b, \ldots)\) occurs in \(H'\). Since another successful \(M.\texttt{chngCurBlock}(b, \ldots)\) follows that one, namely in step \(k\) of \(H\), this contradicts the specification of type \(\tau_{BM}\). \(\square\)

\textbf{Lemma 6.7.} For any history \(H\) of implementation \(I_E\), for any process \(p\), and for any block \(x\), if \(p\) reads \(x \triangleright V\) at line 162 at some point in \(H\), then no process \(q \neq p\) overwrites \(x \triangleright V\) after that point in \(H\).

\textit{Proof.} Recall that a block \(x\) can become current at most once by Lemma 6.5. Next, note that \(x \triangleright V\) is only written at line 173 of \texttt{ECAS}, which occurs before \(x\) becomes current by way of a successful \(M.\texttt{chngCurBlock}(\ldots, x)\) at line 175 (if this ever happens). Thus, \(x \triangleright V\) is never written after \(x\) becomes current, and in particular no process \(q \neq p\) writes \(x \triangleright V\) after \(p\) makes a call to \(M.\texttt{getCurBlock()}\) with response \(x\) at line 160 (hence after \(p\) reads \(x \triangleright V\) at line 162) in \(H\). \(\square\)

To prove linearizability, we now define for each \(H \in \mathcal{H}\) a candidate linearization \(\tilde{H}\) as follows. For each operation execution in \(H\) (complete or pending), we assign a “timestamp”, which is a tuple of the form \((x, t, q)\), where \(x\) is a block address, \(t\) is an integer, and \(q\) is a process ID or zero.

\textbf{Definition 6.8.} The timestamp \(s\) for an arbitrary operation execution \(Op\) in \(H\), say by process \(p\), and its completion (where applicable), are defined as follows:
Chapter 6. Extended Compare-and-Swap

Operation types \text{Read()} or \text{LL()}: 

(a) If \( p \) executes \text{M.getCurBlock()} at line 147 or line 152 during \text{Op}, say with response \( x \), and then reads \( x \triangleright V \) at line 149 or line 154 in step \( i \) of \( H \), then \( s = (x, i, 0) \). (If \text{Op} is pending in \( H \), its completion returns the value read from \( x \triangleright V \).)

(b) Otherwise \( s \) is undefined.

Operation type \text{ECAS}(\text{isSC}, \text{cmp}, \text{new}): 

(c) If \( p \) executes \text{M.getCurBlock()} at line 160 during \text{Op} with response \( x \), reads \( x \triangleright V \) at line 162 in step \( i \) of \( H \), and then executes line 164 or line 166, then \( s = (x, i, 0) \). (If \text{Op} is pending in \( H \), its completion returns the value read from \( x \triangleright V \) and the Boolean assigned to \text{ret} at either line 164 or line 166 is executed.)

(d) Else if \( p \) executes a successful \text{M.chngCurBlock}(b, x) at line 175 during \text{Op} in step \( i \) of \( H \), then \( s = (x, i, 0) \). (If \text{Op} is pending in \( H \), its completion returns the value read from \( b \triangleright V \) and \text{true}.)

(e) Else if \text{Op} is complete in \( H \), and \( p \) executes a failed \text{M.chngCurBlock}(b, y) at line 175 during \text{Op}, then letting \( i \) denote the atomic step (by any process) in \( H \) involving a successful \text{M.chngCurBlock}(b, x), then \( s = (x, i, p) \). (Since \( b \) is a value returned by \text{M.getCurBlock()} and then a failed \text{M.chngCurBlock}(b,...) occurs in \( H \), step \( i \) exists and is uniquely defined by the specification of type \( \tau_{\text{ECAS}} \) and Lemma 6.6. Note also that process IDs are numbered starting at one and so clauses (d) and (e) never yield the same timestamp.)

(f) Otherwise \( s \) is undefined.

It follows from the above definitions that each operation execution for which the timestamp is defined has a unique timestamp. To construct our candidate linearization \( \bar{H} \), we take all operation executions for which the timestamp is defined, and arrange these in increasing order of their timestamp according to Definition 6.9. Operation executions that are pending in \( H \) are completed, as described earlier, if their timestamps are defined, and are discarded from \( \bar{H} \) otherwise.

\textbf{Definition 6.9.} For timestamps \((x_1,t_1,q_1)\) and \((x_2,t_2,q_2)\), we say that \((x_1,t_1,q_1) < (x_2,t_2,q_2)\) if and only if one of the following holds:
• $x_1 \neq x_2$ and block $x_1$ was current before block $x_2$ in $H$

(Note that we can determine which of $x_1$ and $x_2$ was current first by Lemma 6.5.)

• $x_1 = x_2$ and $t_1 < t_2$

• $x_1 = x_2$, $t_1 = t_2$, and $q_1 < q_2$

In this ordering, certain groups of concurrent ECAS operation executions have timestamps that match in the first two positions, say $(x, t, ...)$. The operation execution with timestamp $(x, t, 0)$ is successful; the others, with timestamps of the form $(x, t, p)$ for some process ID $p$, fail. Since their timestamps differ only in the third position, all operation executions in this group appear to take effect at almost the same point in $H$ – at step $t$. That is, the successful ECAS takes effect by applying a successful $\text{chngCurBlock}$ in step $t$, and the others take effect immediately after this. (Note that “behind the scenes”, these failed ECAS operation executions busy-wait, using pseudo-locks, until the successful one has taken effect; see Figure 5.2.)

Next, we define some useful notation. Let $Op_i$, $s_i$, and $p_i$ denote the $i$’th operation execution in $H$ (counting from 1), its timestamp, and the executing process. Also let $(\text{isSC}_i, \text{cmp}_i, \text{new}_i)$ denote the arguments of $Op_i$ in the case when $Op_i$ is an ECAS operation execution. We now prove that $H$ satisfies all the properties of a linearization.

**Lemma 6.10.** $H$ satisfies property (a) of linearizability (sequential completion).

*Proof.* This follows directly from our construction on $H$. In particular, any operation execution that is complete in $H$ has its timestamp defined. \hfill \square

**Lemma 6.11.** If an operation execution $Op$ in $H$ has timestamp $s = (...) t,...$ then $Op$ is pending in state $H[t]$.

*Proof.* The lemma follows immediately from Definition 6.8 unless $Op$ falls under clause (e). In the latter case, during $Op$ there is a call to $M\.getCurBlock()$ with response $b$ followed by a failed $M\.chngCurBlock(b, ...)$, and step $t$ is the one in which a successful $M\.chngCurBlock(b, ...)$ occurs. It follows from the specification of type $\tau_{ECAS}$ and from Lemma 6.6 that this successful $\text{chngCurBlock}$ is unique and occurs in $H$. Moreover, by Lemma 6.5 it must occur between the $\text{getCurBlock}$ and failed $\text{chngCurBlock}$ during $Op$. Thus, $Op$ is pending just after step $t$, as wanted. \hfill \square

**Lemma 6.12.** If an operation execution $Op$ in $H$ has timestamp $s = (x, t, ...)$ then block $x$ is current at some point during $Op$. 
Proof. If \( s \) does not fall under Definition 6.8 (d)–(e) then the call to \texttt{M.getCurBlock()} during \( Op \) returns \( x \), and so \( x \) is current at that point. If \( s \) falls under Definition 6.8 (d) then the process that executes \( Op \) makes \( x \) current in step \( t \), which occurs during \( Op \) by Definition 6.11. If \( s \) falls under Definition 6.8 (e) then \( x \) becomes current in step \( t \) as explained in the proof of Lemma 6.11. \( \square \)

**Lemma 6.13.** \( \bar{H} \) satisfies property (b) of linearizability (order preservation).

**Proof.** Consider two distinct operation executions \( Op_i \) and \( Op_j \) in \( \bar{H} \), with timestamps \( s_i = (x_i, t_i, ...) \) and \( s_j = (x_j, t_j, ...) \) respectively, such that \( Op_i \) precedes \( Op_j \) in \( H \). We must show that \( s_i < s_j \).

**Case A:** \( x_i = x_j \). It follows from Lemma 6.11 that \( Op_i \) and \( Op_j \) are pending in states \( H[t_i] \) and \( H[t_j] \), respectively. Since \( Op_i \) precedes \( Op_j \) in \( H \), this implies that \( t_i < t_j \). Since \( x_i = x_j \) and \( t_i < t_j \), \( s_i < s_j \) by Definition 6.9.

**Case B:** \( x_i \neq x_j \). It follows from Lemma 6.12 that \( x_i \) and \( x_j \) are current at some point during \( Op_i \) and \( Op_j \), respectively. Since we assume \( Op_j \) precedes \( Op_i \) in \( H \) and since \( x_i \neq x_j \), it follows that \( x_i \) becomes current before \( x_j \) in \( H \). (The order in which blocks become current is well-defined by Lemma 6.5.) Thus, \( s_j < s_i \) by Definition 6.9. \( \square \)

**Lemma 6.14.** For any history \( H \) of implementation \( I_E \) where Condition 6.4 holds, any process \( p \), any block \( x \), and any operation execution \( Op_i \) in \( \bar{H} \) by \( p \), if the timestamp \( s_i \) of \( Op_i \) is of the form \((...,0)\), and \( p \) receives response \( v \) when it reads \( x \triangleright Linked[p] \) during the counterpart of \( Op_i \) in \( H \), then \( v = \text{true} \) if and only if there is an LL operation execution in \( \bar{H} \) before \( Op_i \) and after any ECAS that precedes \( Op_i \) in \( \bar{H} \) and returns \((...,\text{true})\).

**Proof.** It follows from the access procedures of \( I_E \), and the initialization of \( x \triangleright Linked[p] \) to \texttt{false}, that \( p \) can only read \texttt{true} from \( x \triangleright Linked[p] \) during \( Op_i \) if it previously completed an LL operation execution during which \( p \) assigned \( x \triangleright Linked[p] = \text{true} \) at line \texttt{156}. It remains to show that this LL occurs after any \texttt{ECAS} that precedes \( Op_i \) in \( \bar{H} \) and returns \((...,\text{true})\). Suppose otherwise. Let \( Op_l \) be \( p \)'s LL, and let \( Op_e \) be the first \texttt{ECAS} that occurs between \( Op_l \) and \( Op_i \) in \( \bar{H} \) and returns \((...,\text{true})\). Since \( Op_l \) is an LL, it follows from Condition 6.4 that \( Op_e \) has argument \( \text{isSC}_e = \text{true} \). Consequently, it follows from our definition of \( Op_e \) and from the \texttt{ECAS} access procedure that a successful \texttt{M.chngCurBlock} occurs during the counterpart of \( Op_e \) in \( H \) that makes some block \( y \) current, where \( y \neq x \) by Lemma 6.5. Now consider the timestamps of \( Op_l, Op_e \), and \( Op_i \). For \( Op_l \), \( s_l \) is of the form \((x, ..., 0)\) by Definition 6.8 (a). For \( Op_e \), \( s_e \) is of the form \((y, ..., 0)\) by
Definition 6.8 (d). Finally, the timestamp \( s_i \) of \( Op_i \) is either of the form \((x, \ldots, 0)\) or of the form \((y', \ldots, 0)\) for some block \( y' \neq x \) by Definition 6.8.

**Case A:** \( s_i \) is of the form \((x, \ldots, 0)\). Since \( s_l \) is of the form \((x, \ldots, 0)\) and \( s_e \) is of the form \((y, \ldots, 0)\), where \( y \neq x \), as noted earlier, Definition 6.9 contradicts \( Op_e \) occurring between \( Op_l \) and \( Op_i \) in \( \bar{H} \).

**Case B:** \( s_i \) is of the form \((y', \ldots, 0)\) for some block \( y' \neq x \). It follows that \( s_i \) falls under Definition 6.8 (d), and so a successful \( M.chngCurBlock(x', y') \) occurs during the counterpart of \( Op_i \) in \( H \) for some \( x' \). In particular, \( x' = x \) by the algorithm and the assumption that \( p \) accesses \( x \triangleright Linked[p] \) during the counterpart of \( Op_i \) in \( H \). Consequently, by Lemma 6.5, \( y' \) is the next block that becomes current after \( x \). But in that case by our construction of \( \bar{H} \), \( Op_i \) and \( Op_e \) are the same operation execution, which contradicts our definition of \( Op_e \). \( \square \)

**Lemma 6.15.** If \( H \) satisfies Condition 6.4 then \( \bar{H} \) satisfies property (c) of linearizability (conformity to type \( \tau_{ECAS} \)).

**Proof.** Let \( H \) be any history of \( I_E \) where Condition 6.4 holds. Since conformity to a type is a safety property it suffices to consider finite \( \bar{H} \). Let \( k = |\bar{H}| \). Let \( x_i \) and \( t_i \) denote the first two components of \( s_i \). Define \( x_0 \) as the initial block, and \( x_{k+1} \) as the current block at the end of \( H \). Define \( s_0 = (x_0, 0, 0) \) and \( s_{k+1} = (x_{k+1}, \infty, 0) \). Let \( \nu_i \) for \( 0 \leq i \leq k \) denote the state of a correctly implemented ECAS object after applying the first \( i \) operation executions in \( \bar{H} \). Let \( \nu_i.V \) and \( \nu_i.Linked[1..N] \) denote the two components of this state.

We will show that for any \( i \in \mathbb{N}, 0 \leq i \leq k \):

(a) For \( t = t_i \) and any integer \( t \in [t_i, t_{i+1}) \), \( x_i \triangleright V = \nu_i.V \) holds in state \( H[t] \).

(b) If \( i > 0 \), then the response of \( Op_i \) is the correct response for an operation execution of that type applied in state \( \nu_{i-1} \).

Part (b) implies the lemma, but we require both parts for induction. Now let \( S(i) \) denote parts (a)–(b) for a particular value of \( i \). Note that in \( H \), the current block and value of field \( V \) in that block are changed only by an execution of line 175, which is an atomic step that defines the timestamp of an operation execution (on the target object) in \( \bar{H} \). Therefore, the current block and value of \( V \) in that block do not change between atomic steps \( t_i \) and \( t_{i+1} \) in \( H \). This, in turn, implies that to prove part (a) of \( S(i) \), it suffices to prove that \( x_i \triangleright V = \nu_i.V \) in state \( H[t_i] \) – and that is all we do in the inductive step that follows.
For $S(0)$, part (a) follows from our earlier definition of $x_0$ as the initial block and $t_0 = 0$, as well as the initialization of $x_0 \triangleright V$ to the initial value of type $\tau_{ECAS}$. Part (b) holds trivially for $S(0)$. Now for any $i$, $0 < i \leq k$, suppose that $S(i - 1)$ holds, and consider $S(i)$. We proceed by cases on how $s_i = (x_i, t_i, ...) was obtained, noting that $x_i = x_{i-1}$ unless $s_i$ falls under Definition 6.8 (d).

**Case A:** $Op_i$ falls under Definition 6.8 (a)–(b). In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, and either $Op_i$ is a Read and $p_i$ reads $x \triangleright V$ in step $t_i$ of $H$ at line 149, or $Op_i$ is an LL and $p_i$ reads $x \triangleright V$ in step $t_i$ of $H$ at line 154.

$S(i)$ (a) follows from $S(i - 1)$ (a) because $\nu_i = \nu_{i-1}$, $x_i = x_{i-1}$, and since step $t_i$ by $p_i$ does not change the current block or write $x_{i-1} \triangleright V$.

For $S(i)$ (b), note that by the algorithm, $Op_i$ returns the value $p$ reads from $x_i \triangleright V$ in step $t_i$, which equals $\nu_{i-1}.V$ by $S(i - 1)$ (a) and the fact that $x_{i-1} = x_i$. Since $Op_i$ is a Read or LL, this response is correct in $\bar{H}$.

**Case B:** $Op_i$ falls under Definition 6.8 (c), and $p_i$ executes line 164 during the counterpart of $Op_i$ in $H$. In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, and $Op_i$ is an ECAS operation execution where $p_i$ reads $x_i \triangleright V$ at line 162 in step $t_i$ of $H$.

As in Case A, it follows that $p_i$ reads the value $\nu_{i-1}.V$ from $x_{i-1} \triangleright V$ at line 162. Similarly, it follows from Lemma 6.14 that $p_i$ reads the value $\nu_{i-1}.Linked[p_i]$ from $x_{i-1} \triangleright Linked[p]$ at line 163. Consequently, $Op_i$ returns $(\nu_{i-1}.V, false)$ in $\bar{H}$. Furthermore, by the success of the test at line 163, either $(isSC = true \land \nu_{i-1}.Linked[p_i] = false)$ holds, or $(isSC = false \land cmp_i \neq \nu_{i-1}.V)$ holds. Thus, $Op_i$ fails.

$S(i)$ (a) follows from $S(i - 1)$ (a) as in Case A since $Op_i$ is a failed ECAS operation execution, and so $\nu_i.V = \nu_{i-1}.V$.

$S(i)$ (b) holds since $Op_i$ returns $(\nu_{i-1}.V, false)$ in $\bar{H}$ and since $Op_i$ is a failed ECAS operation execution.

**Case C:** $Op_i$ falls under Definition 6.8 (c), and $p_i$ executes line 166 during the counterpart of $Op_i$ in $H$. In this case, $s_i = (x_i, t_i, 0)$ for some $t$, and $Op_i$ is an ECAS operation execution where $p_i$ reads $x_i \triangleright V$ at line 162 in step $t$ of $H$.

As in Case B, it follows that $p_i$ reads the value $\nu_{i-1}.V$ from $x_{i-1} \triangleright V$ at line 162, and the value $\nu_{i-1}.Linked[p]$ from $x_{i-1} \triangleright Linked[p]$ at line 163. Consequently, $Op_i$ returns $(\nu_{i-1}.V, true)$ in $\bar{H}$. Furthermore, by the success of the test at line 165,
isSC = false, cmp_i = new_i, and cmp_i equals the value read from x_{i-1} ∆ V, which is ν_{i-1}.V. Thus, Op_i is successful, and leaves ν_i.V = ν_{i-1}.V.

S(i) (a) follows from S(i-1) (a) because because Op_i is a successful ECAS operation execution where ν_i.V = ν_{i-1}.V, and since step t_i by p_i does not change the current block or write x_{i-1} ∆ V.

S(i) (b) holds since Op_i returns (ν_{i-1}.V, true) in H and since Op_i is a successful ECAS operation execution.

Case D: Op_i falls under Definition 6.8 (d), and p_i executes a successful M.chngCurBlock(d, d') for some d and d' at line 175 during the counterpart of Op_i in H. In this case, s_i = (x_i, t_i, 0) for some t_i, x_i = d', x_{i-1} = d (by Lemma 6.6) and Op_i is an ECAS operation execution.

Since x_{i-1} = d, it follows that p_i’s call to M.getCurBlock() during the counterpart of Op_i in H returns x_{i-1}, and so as in Case B p_i reads the value ν_{i-1}.Linked[p] from x_{i-1} ∆ Linked[p]. Similarly, it follows from Lemma 6.7 and the algorithm that the value p_i reads from x_{i-1} ∆ V is the value of this shared variable in state H[t_i - 1], which is ν_{i-1}.V by S(i-1) (a). Consequently, Op_i returns (ν_{i-1}.V, true) in H. Furthermore, by the failure of the tests at line 163 and line 165, it follows that either (isSC = false ∧ cmp_i = ν_{i-1}.V ∧ cmp_i ≠ new_i) holds, or (isSC = true ∧ ν_i.Linked[p_i] = true) holds. In either case, Op_i is successful.

S(i) (a) follows by the action of step t_i by p_i, which is a M.chngCurBlock() that makes block x_i current, and where x_i = new_i by p_i’s prior execution of line 173. (It follows from the algorithm and Specification 5.1 that no process can overwrite this value until possibly after step t_i.)

S(i) (b) holds since Op_i returns (ν_{i-1}.V, true) in H and since Op_i is a successful ECAS operation execution.

Case E: Op_i falls under Definition 6.8 (e), and p_i executes a failed M.chngCurBlock(d, ...) for some d at line 175 during the counterpart of Op_i in H. In this case, s_i = (x_i, t_i, p_i) for some t_i and Op_i is an ECAS operation execution. Furthermore, there is an ECAS operation execution Op_j in H that falls under Definition 6.8 (d), and whose timestamp is s_j = (x_j, t_j, 0) where x_j = x_i, t_j = t_i, and j < i. Moreover, p_j applies a successful M.chngCurBlock(d, x_j) during the counterpart of Op_j, in step t_j of H.
Now consider the response of $Op_i$. Since $p_i$ completes line 178 during $Op_i$, note that $p_i$’s failed $M.chngCurBlock(d, ...)$ at line 175 returns $p_j$’s ID. Next, $p_i$ reads $new_j$ from $x_i > NextVal[p_j]$ at line 177 (since this read occurs after $p_j$’s successful $chngCurBlock$). Since $Op_j$ is a successful ECAS operation execution (see Case D), $new_j = \nu_j.V$ holds, and so $Op_i$ returns $(\nu_j.V, false)$ in $H$.

Next, we will show that $Op_i$ fails and $\nu_{i-1}.V = \nu_j.V$. The latter point follows from $S(i-1)$ (a) and Definitions 6.8 and 6.9, which imply that either $j = i - 1$, or the only operation executions between $Op_j$ and $Op_i$ in $H$ are ones with timestamps of the form $(x_j, t_j, ...)$ (and fall under Definition 6.8 (e), like $Op_i$). It remains to show that $Op_i$ fails. If $isSC_i = true$, then this holds because $Op_j$ is the last successful ECAS operation execution in $H$ before $Op_i$ and there is no LL operation execution by $p_i$ between $Op_j$ and $Op_i$ in $H$ (by Definitions 6.8 and 6.9), and so $\nu_{i-1}.Linked[1..N] = false$. Otherwise, $isSC_i = isSC_j = false$ (by Condition 6.4), and so $cmp \neq new$ holds for both $Op_i$ and $Op_j$ by the failure of the test at line 165. It suffices to show that $cmp_i \neq \nu_{i-1}.V$. To that end, since $\nu_{i-1}.V = \nu_j.V = new_j$ and $cmp_j \neq new_j$, it suffices to show that $cmp_i = cmp_j$. To show this, it suffices in turn to show that $p_i$ and $p_j$ read the same value from $d>V$, since this value equals $cmp$ in both operation executions by the failure of the test at line 163. Suppose otherwise. (Once again, we give a general argument that is slightly more complex than one needed to deal with implementation $I_E$ alone, but is used later on in Chapter 7.2.)

Then some process $q$ writes $d>V$ between $p_i$ and $p_j$ reading it. It follows that $q$ is not $p_i$ or $p_j$, because neither $p_i$ nor $p_j$ writes $d>V$ during the counterparts of $Op_i$ and $Op_j$ (respectively) in $H$, and neither $p_i$ nor $p_j$ accesses block $d$ again after completing $Op_i$ and $Op_j$ by Lemma 6.5, $p_j$’s successful $chngCurBlock(d, ...)$, and $p_i$’s failed $chngCurBlock(d, ...)$. However, $q \notin \{p_j, p_i\}$ contradicts Lemma 6.7.

Thus $Op_i$ fails and leaves $\nu_i.V = \nu_{i-1}.V$. $S(i)$ (a) follows from $S(i-1)$ (a) since $\nu_{i-1}.V = \nu_j.V$ holds and $t_j = t_i$. $S(i)$ (b) holds because $Op_i$ returns $(\nu_{i-1}.V, false)$ in $H$ and since $Op_i$ is a failed ECAS operation execution.

\[ \square \]

**Theorem 6.16.** The implementation $I_E$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the CC and DSM models.
Proof. Specification 6.1 under Condition 6.4 follows directly from Lemma 6.10, Lemma 6.13, and Lemma 6.15. Specification 6.2 follows from the structure of the access procedures and from Specification 5.2. RMR complexity follows from the structure of the access procedures, the RMR complexity of the base objects, including the block manager object $M$, and the RMR complexity of subroutine $\text{AllocBlock}()$.

\[\square\]

6.2 Conclusion

In this chapter we presented an $O(1)$-RMR implementation of a shared object type ECAS that can simulate either CAS or LL/SC, but not both “simultaneously” (see Condition 6.4). Thus, we have proved part of our principal result (1) stated in Chapter 1. (We discuss how to implement arbitrary comparison primitives in Chapter 9.)

On first impression it might appear that given such implementations, we can take any algorithm that uses reads, writes and CAS, and “simulate” it by an algorithm that uses reads and writes only, with at most a constant-factor increase in RMR complexity. (The “simulation” would replace any shared object on which operation type CAS is applied with our software implementation.) However, this conclusion does not follow because our implemented CAS object (and similarly for LL/SC) is weaker than its hardware counterpart in two important ways. First, $O(1)$ RMR complexity is not enough. For example, in a cache-coherent multiprocessor with write-back caching, any shared object can be held in-cache and accessed locally, whereas every successful CAS operation (except one where $\text{cmp} = \text{new}$) in our implementations incurs several RMRs. Similarly, in a DSM multiprocessor a shared object can be accessed locally by some processor, whereas in our implementations every process may incur RMRs executing certain operations. Second, our implementation of CAS (as presented so far) does not support the Write operation, whereas in hardware both Write and CAS (if provided) can be applied to any shared object. We deal with the first issue in Chapter 7 and with the second issue in Chapter 8. We then assert in Chapter 9 our principal result (3) from Chapter 1.
Consider an algorithm $A$ that accesses an ECAS object $E$. Our goal in this chapter is to show that one can “plug” our simulation of $E$ into $A$ with at most a constant-factor increase in $A$’s RMR complexity, and while preserving other important correctness properties. (See Chapter 9 for a discussion of these properties.) To achieve this, our implementation of $E$ must meet the following locality properties, in addition to worst-case $O(1)$ RMR complexity: (1) in the DSM model, a designated process $p_{\text{special}}$ (to which we say $E$ is local) may execute any operation execution on $E$ without incurring any RMRs; and (2) in the CC model, certain “in-cache” operation executions on $E$ incur no RMRs.

Since the first property above is straightforward to define, we now focus on formalizing the second property. To begin with, we must define precisely when an operation on a shared object provided in hardware is an RMR in the CC model. Our definition is very much tied in with the one given in Chapter 2 where the only primitives considered for accessing shared memory were reads and writes. In a multiprocessor that supports other types of shared objects in hardware, we must determine which operations cause RMRs and how many. (Recall from Chapter 2 that an operation is the application of an operation type to a shared object, and comes in two flavours: atomic operations represented by atomic steps and non-atomic operations represented by operation executions.) To that end, we classify operations of every shared object type as being either read-like or write-like, and apply our definition from Chapter 2 by treating read-like operations like reads and write-like operations like writes. (We classify operations and not operation types because two applications of the same operation type may behave differently.) But what should we classify as read-like and write-like? To a first approximation, an opera-
tion is write-like if it changes the state of the shared object, and read-like otherwise. One has to be careful about the interpretation of this statement, however, because in certain cases there is more than one natural way to classify a particular operation.

Consider an object of type $\tau_{CAS}$ (as defined in Chapter 6). For this type, it is natural to treat $\text{Read}$ and failed $\text{CAS}$ as read-like, and successful $\text{CAS}$ as write-like. However, in the special case when the arguments of a $\text{CAS}$ operation satisfy $\text{cmp} = \text{new}$, it is also natural to classify a successful $\text{CAS}$ as read-like because the state transition it causes is trivial. Given such a choice, the safe thing to do is to classify the operation as read-like, because the more operations are classified as read-like, the stronger become the correctness properties of locally-accessible implementations, and the harder it becomes to construct them. In particular, the more operations are read-like, the fewer RMRs the access procedures of the implementation are permitted to incur. (In this particular case, we can also assume without loss of generality that $\text{CAS}$ is never called with $\text{cmp} = \text{new}$, since such calls can be simulated using $\text{Read}$.)

Next, consider type $\tau_{LL/SC}$ (as defined in Chapter 6). Recall that, according to the definition in Chapter 6, this state consists of a value denoted $V$ and a Boolean array $\text{Linked}[1..N]$. Applying our convention to this definition we classify failed $\text{SC}$ as read-like, successful $\text{SC}$ as write-like, and $\text{LL}$ as write-like. However, note that when process $p$ applies $\text{LL}$, despite writing $\text{Linked}[p]$ this operation has no effect on the component of state that is read by other processes because only $p$ reads $\text{Linked}[p]$. Consequently, the change of state need not be propagated by the coherence protocol, and an RMR need not occur unless the value $V$ is also loaded into the local cache. (A simple hardware implementation of $\text{LL/SC}$ encodes $\text{Linked}[p]$ in the state of $p$’s cached copy of the shared object. An $\text{LL}$ by $p$ sets this bit only in $p$’s cache, and a successful $\text{SC}$ triggers an invalidation that clears this bit in all caches holding copies of the shared object.) Thus, it is also natural to treat $\text{LL}$ as read-like, and once again it is safe for us to do so as it increases the burden on the implementation. (Another reason for treating $\text{LL}$ as read-like is that another definition of $\tau_{LL/SC}$ exists where $\text{LL}$ does not modify the shared state at all.)

Finally, consider type $\tau_{ECAS}$, which we defined in Chapter 6 to serve as a precursor for $\text{CAS}$ and $\text{LL/SC}$. We classify operations on such an object as follows: $\text{Read}$, $\text{LL}$, and failed $\text{ECAS}$ are read-like, and successful $\text{ECAS}$ is write-like except when $\text{isSC} = \text{false}$ and $\text{cmp} = \text{new}$. (In the latter case, there is more than one natural classification, but we are free to classify such operations as read-like, as explained earlier.)

Now consider a (linearizable) implementation $I$ of a shared object type $\tau$ using only
reads and writes. We would like the implementation \( I \) to have the following \textit{RMR-preservation property} in the write-through or write-back CC model:

**Definition 7.1** (RMR-preservation property in the CC model). There exist positive constants \( c_1, c_2 \) such that for any history \( H \) of the implementation \( I \) there is a linearization \( \tilde{H} \) of \( H \) such that for any process \( p \), if \( p \) incurs \( X \) RMRs in \( H \) in the particular CC model under consideration and \( Y \) RMRs in \( \tilde{H} \) in the same model, then \( X \leq c_1 Y + c_2 \).

Informally, Definition 7.1 states that the RMR cost of \( H \) for any process \( p \) is at most a constant factor greater than the RMR cost of \( \tilde{H} \). Consequently, replacing a shared object with one implemented using \( I \) causes at most a constant-factor increase in RMR complexity in any history. This constant factor is \( c_1 \). The term \( c_2 \) is needed to compensate for any pending operation execution by \( p \) in \( H \) that may be discarded from \( \tilde{H} \). In particular it is possible that \( \tilde{H} \) is empty when \( H \) is not empty, in which case \( Y = 0 \) and \( X > 0 \). (In that case, \( X \in O(1) \) because \( H \) contains at most one operation execution by \( p \) and because we assume \( I \) has \( O(1) \) worst-case RMR complexity).

An implementation that merely satisfies \( O(1) \) RMR complexity per operation does not automatically satisfy Definition 7.1. This is because in the worst case \( X \) grows linearly with the number of operations invoked on the target object in \( H \), and yet \( Y \) may grow much less quickly because many of these operations are applied “in-cache”. In particular, in the write-back CC model if only one process is active in \( H \) then \( Y = O(1) \) no matter how many operations that process applies on the target object.

We now state conditions on the implementation \( I \) that are sufficient for it to satisfy this RMR preservation property in the write-through and write-back CC model. We call these “locality properties”. (We discuss the necessity and sufficiency of these conditions in Section 7.4 below.)

For the write-through CC model, the locality property informally states that any process \( p \) can apply multiple operation executions “in-cache” (i.e., on a “shared cached copy” of the object) provided these operation executions are read-like, and not interleaved with any write-like operation executions. More formally, we have:

**Definition 7.2** (locality property for the write-through CC model). Let \( O_\tau \) denote the target object implemented by \( I \). For any history \( H \) of \( I \), there is a linearization \( \tilde{H} \) of \( H|O_\tau \) such that the following property holds:

\((\text{R})\)

For any process \( p \), if \( \tilde{H}' \) is a contiguous subsequence of \( \tilde{H} \) consisting only of read-
like operation executions on $O_\tau$, and $H'$ is the subsequence of corresponding base object atomic steps in $H$, then $p$'s steps in $H'$ cause only a constant number of RMRs in $H$.

For the write-back CC model, the locality property builds on Definition 7.2. Informally, it states that (in addition to the above) any process $p$ can apply multiple operation executions “in-cache” (i.e., on an “exclusive cached copy” of the object) provided these operation executions are not interleaved with any operation executions by other processes. More formally, we have:

**Definition 7.3** (locality property for the write-back CC model). Let $O_\tau$ denote the target object implemented by $I$. For any history $H$ of $I$, there is a linearization $\bar{H}$ of $H|O_\tau$ such that property (R) stated in Definition 7.2 holds, and furthermore the following property holds:

 (**W**) For any process $p$, if $\bar{H}'$ is a contiguous subsequence of $\bar{H}$ consisting only of operation executions issued by $p$ on $O_\tau$, and $H'$ is the subsequence of corresponding base object atomic steps in $H$, then $p$'s steps in $H'$ cause only a constant number of RMRs in $H$.

In the remainder of this chapter we present locally-accessible implementations of CAS, LL/SC, and ECAS. Note that we will use two notions of locality in our analysis – the locality of the target object and the locality of the base objects used to construct the target object. To prove that an implementation $I$ satisfies a locality property, we will first apply Definitions 7.2 and 7.3 to define our burden of proof with respect to a given history $H$ of $I$. Then, to construct such a proof, we will appeal to the RMR complexity and locality of the base objects accessed in $H$. To that end, we will consider sequences of atomic steps where some $O(1)$-RMR base object $B$ is accessed, and apply the analogs properties (R) and (W) above to those sequences. (The analog of (R) states that process $p$ incurs $O(1)$ RMRs accessing a base object $B$ in any contiguous subsequence of $H$ where no write-like operation is applied to $B$. The analog of (W) states that process $p$ incurs $O(1)$ RMRs accessing a base object $B$ in any contiguous subsequence of $H$ where only $p$ accesses $B$.)

### 7.1 Locally-Accessible CAS and LL/SC

To illustrate the techniques used for proving locality properties, we first consider the simple implementations of CAS and LL/SC from ECAS presented at the beginning of
Theorem 7.4. The implementations of CAS and LL/SC presented in Figure 6.4 satisfy the locality property in the DSM model and the two types of CC model under consideration, provided that the implementation of the ECAS base object $B$ satisfies the same locality property.

Proof. Because in these implementations of CAS and LL/SC each access procedure applies exactly one atomic step on the base object $B$, the RMR cost of executing any access procedure equals the RMR cost of the corresponding atomic step on $B$. Now suppose that the implementation of $B$ satisfies the locality property in one of the models under consideration.

DSM model. The locality property follows directly from the above observation because if $B$ satisfies the locality property in the DSM model with respect to some designated process $p_{\text{special}}$, then each operation execution applied by $p_{\text{special}}$ on the target object incurs zero RMRs.

CC model. Recall that operation executions on the target objects are linearized at the point when the corresponding operation executions on $B$ occur, as discussed in Chapter 6. Now consider a history $H$ of the implementation, and let $\bar{H}$ be this particular linearization of $H|O_\tau$, where $O_\tau$ is the target object.

For property (R), fix $p$ and $\bar{H}'$ as in Definition 7.2, and consider $H'$. Since $\bar{H}'$ consists of read-like operation executions only, and since in Figure 6.4 a read-like operation execution on the target object only applies a read-like operation on the ECAS base object, it follows that $H'$ is a contiguous subsequence of atomic steps in $H$ that apply read-like operations on $B$. Consequently, by the locality property (R) of $B$’s implementation, $p$ incurs $O(1)$ RMRs in $H$ applying its steps from $H'$, as wanted.

The proof for property (W) is very similar. Fix $p$ and $\bar{H}'$ as in Definition 7.3, and consider $H'$. Since $\bar{H}'$ consists of operation executions by $p$ only, and since in Figure 6.4 an operation execution on the target object applies exactly one operation on the ECAS base object (at which point it takes effect), it follows that $H'$ is a contiguous subsequence of atomic steps in $H$ where only $p$ accesses $B$. Consequently, by the locality property (W) of $B$’s implementation, $p$ incurs $O(1)$ RMRs in $H$ applying its steps from $H'$, as wanted. 

\qed
Chapter 7. Locally-Accessible Implementations

7.2 Locally-Accessible ECAS for the CC Model

The implementation of ECAS presented in Chapter 6 satisfies the locality property in the write-through model (Definition 7.2) when the block manager is implemented as described in Chapter 5. The locality property for the write-back model (Definition 7.3) does not hold, however, because a process performing \( k \) successful ECAS operation executions in a solo history will incur \( \Omega(k) \) RMRs as it accesses \( k \) different blocks. (Property (W) in Definition 7.3 requires that only \( O(1) \) RMRs occur in such a history.)

In the remainder of this chapter, we show how to modify our earlier implementation of ECAS to achieve the locality property in the write-back CC model. (As we show later, the same modifications yield the locality property in the write-through CC model.) The high-level idea is to allow a process to perform multiple write-like operation executions on an uncontended ECAS object while accessing only \( O(1) \) blocks. To that end, a process \( p \) executing a write-like (i.e., successful ECAS) operation execution will attempt to reuse the current block instead of allocating a new one. The main challenge here is to handle correctly the race condition when \( p \) attempts to reuse some block \( x \) while some other process \( q \) attempts to access block \( x \).

The modifications to the implementation from Chapter 6 are twofold: First, we use a \( O(1) \)-RMR block manager that satisfies the locality property for the write-through CC model (see Definition 7.2). (This is sufficient even in the write-back CC model.) The block manager implementation from Chapter 5 can be used for this purpose, as we show later on (see Lemma 7.17). Second, we override the subroutines \texttt{HelperBegin}, \texttt{HelperEnd} and \texttt{HelperCC} with the implementations shown in Figure 7.1. To support these subroutines, we alter the structure of a block by introducing several new fields. Boolean flags \texttt{seen}, \texttt{accessed}, and \texttt{changing} are used to keep track of whether a block has been accessed, whether an operation execution that accessed the block has taken effect, and whether the block is being reused (respectively). (In this context we refer to accesses to a block that occur outside of the block manager.) These three flags are needed for the locality property and their use is explained in the next paragraph. In addition, Boolean arrays \texttt{seenBy}[1..N] and \texttt{accessedBy}[1..N] are used to record which process has written \texttt{seen} and \texttt{accessed}, respectively. Statements that access these arrays are shaded to indicate that they are needed only for RMR complexity (i.e., to reduce RMRs incurred while accessing \texttt{seen} and \texttt{accessed}, as we explain later). Thus, when we reason about linearizability, these statements can be effectively ignored.
The new subroutines work as follows. Recall first that for any block \( x \), the field \( \text{writer} \) stores the ID of the process that allocated the block, or \( \perp \) for the initial block. Whenever some process \( p \) applies an operation execution on an ECAS object where it accesses some block \( x \), and \( p \neq x \triangleright \text{writer} \), \( \text{HelperBegin}(x) \) announces (to the process whose ID is recorded in \( x \triangleright \text{writer} \)) that block \( x \) has been accessed by another process by ensuring that \( x \triangleright \text{seen} = \text{true} \). Similarly, \( \text{HelperEnd}(x) \) announces that such an operation execution has taken effect by ensuring that \( x \triangleright \text{accessed} = \text{true} \). Function \( \text{HelperBegin}(x) \) also waits at line 191 for process \( x \triangleright \text{writer} \) to finish reusing block \( x \), if it has already started to do so. Thus, process \( x \triangleright \text{writer} \) can use \( x \triangleright \text{seen} \) to decide when it can reuse block \( x \) safely, and can use \( x \triangleright \text{accessed} \) to decide when it can perform additional RMRs without jeopardizing the locality property. Function \( \text{HelperCC}(x, \text{new}) \) allows process \( x \triangleright \text{writer} \), under certain conditions, to access \( x \triangleright V \) in mutual exclusion, and hence to apply a successful ECAS operation execution by reusing block \( x \) rather than allocating a new one. The subroutines \( \text{TryToReuseBlock}(x) \) and \( \text{DoneReusingBlock}(x) \), which are called from \( \text{HelperCC}(x, \text{new}) \), set and reset \( x \triangleright \text{changing} \) to announce that block \( x \) is being reused, and should not be accessed during that time.

The subroutines \( \text{HelperBegin}(x) \) and \( \text{HelperEnd}(x) \) use the arrays \( x \triangleright \text{seenBy}[1..N] \) and \( x \triangleright \text{accessedBy}[1..N] \) in addition to \( x \triangleright \text{seen} \) and \( x \triangleright \text{accessed} \) only to meet RMR complexity bounds imposed by locality property (R) (see Definition 7.2). To see why these arrays are needed, consider a history where processes execute only failed ECAS operation executions on the target object, in which case Definition 7.2 requires that each process incur \( O(1) \) RMRs in the entire history. If the shaded statements that access \( \text{seenBy} \) and \( \text{accessedBy} \) were not present, processes could incur arbitrarily many RMRs writing \( \text{seen} \) and \( \text{accessed} \) in the write-through CC model. Even if the writes at line 189 and line 196 were preceded by tests checking whether \( \text{seen} \) and \( \text{accessed} \) (respectively) are already \text{true}, a process could incur \( N - 1 \) RMRs executing each test (in the write-through or write-back CC model) as every other process writes \( \text{seen} \) and \( \text{accessed} \) once. (We return to this issue in the proof of Lemma 7.14, Cases D and E.)

7.2.1 Analysis

Let \( I_E \) denote the implementation of ECAS presented in Chapter 6, and let \( I'_E = (\tau_{ECAS}, \mathcal{P}, \mathcal{B}, \mathcal{H}) \) denote the implementation obtained by transforming \( I_E \) as described earlier (i.e., replace the block manager object \( M \) with one that satisfies the locality
Declarations

**Shared variables**: (per-block)
- seenBy[1..N], accessedBy[1..N] – array of Boolean, initially all false
- seen, accessed, changing – Boolean, initially false

**Private variables**: (per-process)
- ret – Boolean, uninitialized

---

Function HelperBegin(d)

**Input**: d – block address

```
186 if read(d ⊲ writer) ≠ PID then
187     if read(d ⊲ seenBy[PID]) = false then
188         write d ⊲ seenBy[PID] := true
189         write d ⊲ seen := true
190     end
191     await d ⊲ changing = false
192 end
```

Function HelperEnd(d)

**Input**: d – block address

```
193 if read(d ⊲ writer) ≠ PID then
194     if read(d ⊲ accessedBy[PID]) = false then
195         write d ⊲ accessedBy[PID] := true
196         write d ⊲ accessed := true
197     end
198 end
```

Function HelperCC(d, new)

**Input**: d – block address
**Input**: new – value to be written in block d
**Output**: Boolean success indicator (true if block d was successfully reused, false otherwise)

```
199 ret := false
200 if read(d ⊲ writer) = PID then
201     if TryToReuseBlock(d) = true then
202         write d ⊲ V := new
203         write d ⊲ Linked[PID] := false
204         ret := true
205     end
206     DoneReusingBlock(d)
207 end
208 return ret
```

Function TryToReuseBlock(d)

**Input**: d – block address
**Output**: Boolean

```
209 write d ⊲ changing := true
210 if read(d ⊲ seen) = false then
211     return true
212 else
213     return false
214 end
```

Function DoneReusingBlock(d)

**Input**: d – block address

```
215 write d ⊲ changing := false
216 if read(d ⊲ seen) = true then
217     await d ⊲ accessed = true
218 end
```

---

Figure 7.1: Subroutines for locally-accessible ECAS implementation in CC model.
property in the write-through CC model, and implement subroutines HelperBegin, HelperEnd, and HelperCC as shown in Figure 7.1). We now establish the correctness properties of $I'_E$.

**Lemma 7.5.** The analog of Lemma 6.5 for $I'_E$ holds.

**Proof.** This follows by the same proof as given in Chapter 6.1. □

**Lemma 7.6.** The analog of Lemma 6.6 for $I'_E$ holds.

**Proof.** This follows by the same proof as given in Chapter 6.1, with any reference to Lemma 6.5 replaced by a reference to Lemma 7.5. □

**Lemma 7.7.** For any history $H$ of $I'_E$, and for any block $x$ accessed in $H$, if process $p$ allocated block $x$ (i.e., $x \triangleright writer = p$) then:

(a) If $p$ has completed a call to TryToReuseBlock($x$) with response true, but has not subsequently made a call to DoneReusingBlock($x$), then no process $q \neq p$ has completed a call to HelperBegin($x$).

(b) If $p$ has completed a call to TryToReuseBlock($x$) with response false, then some process $q \neq p$ has made a call to HelperBegin($x$). Moreover, if $p$ subsequently completes a call to DoneReusingBlock($x$), then some process $q' \neq p$ has made a call to HelperEnd($x$).

(c) If $p$ has completed a call to TryToReuseBlock($x$) with response false during some operation execution $Op$ on the target object, then $p$ does not access block $x$ after completing $Op$.

**Proof.**

**Part (a):** Note that $x \triangleright changing = true$ holds from the moment $p$ completes line 209 during its last call to TryToReuseBlock($x$) until the point in $H$ under consideration. Consequently, if $q$ completed a call to HelperBegin($x$), then it must have completed line 191 before $p$ last executed line 209. But in that case $q$’s execution of lines 187–190 precedes $p$’s last execution of line 210. It then follows by the algorithm that $x \triangleright seen = true$ holds when $p$’s executes line 210. This contradicts $p$ executing line 211.

**Part (b):** If $p$ has completed a call to TryToReuseBlock($x$) with response false then it read $d \triangleright seen = true$ at line 210, which implies that some process $q \neq p$ previously executed line 189, and hence $q$ made a call to HelperBegin($x$). Since $x \triangleright seen = true$
is a stable property, if $p$ subsequently completes a call to \texttt{DoneReusingBlock}(x) then it executes line 217, and does not complete that line until some process $q' \neq p$ executes line 196 of \texttt{HelperEnd}(x).

**Part (c):** If \texttt{TryToReuseBlock}(x) returns false, then the calling function \texttt{HelperCC} also returns false. Furthermore, $p$ executes $M\_chngCurBlock(x, y)$ at line 175 (see Figure 6.6) for some block $y$ during its ECAS operation execution under consideration before it executes another operation execution on the target object. Once this \texttt{chngCurBlock} occurs, $x$ is no longer the current block by Lemma 7.5. Consequently, once $p$ completes $Op$, it never accesses block $x$ again.

**Lemma 7.8.** The analog of Lemma 6.7 for $I_E'$ holds.

**Proof.** The proof of Lemma 6.7 given in Chapter 6.1 breaks, even after replacing references to Lemma 6.5 by references to Lemma 7.5. This is because inside function \texttt{HelperCC}(d, new), a process may write $d \triangleright V$ at line 202 after $d$ has become current. To fix the proof, we must show that for any block $x$, once some process $q$ has read $x \triangleright V$ at line 162, no process $p \neq q$ overwrites $x \triangleright V$ at line 202. Suppose otherwise, and note that $p = x\triangleright writer$ by the test at line 200. Furthermore, when $p$ is at line 202, $q$ has completed a call to \texttt{HelperBegin}(x), $p$ has completed a call to \texttt{TryToReuseBlock}(x) with response true (at line 201), and $p$ has not subsequently called \texttt{DoneReusingBlock}(x) (at line 206). But this contradicts Lemma 7.7 (a).

To prove linearizability, we define for any history $H \in \mathcal{H}$ a candidate linearization $\bar{H}$ as in Chapter 6.1, except that we augment the definition of timestamps (Definition 6.8). That is, we add a new clause (between clause (e) and clause (f)) for an ECAS($isSC, cmp, new$) operation execution $Op$ in $H$ where line 202 is reached:

\begin{itemize}
  \item[(g)] Else if $p$ executes $M\_getCurBlock()$ at line 160 during $Op$, say with response $x$, and then writes $new$ to $x \triangleright V$ at line 202 of \texttt{HelperCC} during $Op$ in step $i$ of $H$, then $s = (x, i, 0)$.
  
  (If $Op$ is pending in $H$, its completion returns the value read from $x \triangleright V$ at line 162 and true.)
\end{itemize}

We now prove key properties of $\bar{H}$, as in Chapter 6.1.

**Lemma 7.9.** The analogs of Lemma 6.10 and Lemma 6.13 for $I_E'$ hold.
Proof. Both lemmas follow for $I_E'$ by the same proofs as in Chapter 6.1, with reference to Lemmas 6.5 and 6.6 replaced by references to Lemmas 7.5 and 7.6.

**Lemma 7.10.** The analog of Lemma 6.14 for $I_E'$ holds.

**Proof.** We modify the proof of Lemma 6.14 given in Chapter 6.1 as follows. First, we replace references to Lemma 6.5 by references to Lemma 7.5. Second, we must consider the case when $O_p$ or $O_i$ is an ECAS operation execution whose timestamp falls under clause (g) above.

If $O_i$ falls under clause (g) and $O_p$ does not, then we deal with $O_i$ in the same way as when $O_i$ is a Read or LL. (See Case A in the proof of Lemma 6.14.)

If $O_p$ falls under clause (g), then it must be that the same process (i.e., $p$) applies $O_i$ and $O_p$. To see this, note that since $O_p$ is the first successful ECAS that occurs between $O_i$ and $O_p$ in $H$, it follows from Definitions 6.8 and 6.9 (as augmented in this chapter) that the $M$.getCurBlock() in the counterparts of both $O_i$ and $O_p$ in $H$ returns $x$. Furthermore, since $O_i$ occurs before $O_p$ in $H$, the call to HelperBegin($x$) during the counterpart of $O_i$ in $H$ occurs before the call to DoneReusingBlock($x$) during the counterpart of $O_p$ in $H$, and so Lemma 7.7 part (a) implies that the same process executes $O_i$ and $O_p$. Thus, $p$ applies $O_i$, and then $O_p$, which assigns $x \triangleright Linked[p] = false$ at line 203. Since $p$ does not apply an LL between $O_p$ and $O_i$, it follows from the algorithm that $p$ reads $x \triangleright Linked[p] = false$ during the counterpart of $O_i$ in $H$, which contradicts the definition of $O_i$.

**Lemma 7.11.** The analog of Lemma 6.15 for $I_E'$ holds.

**Proof.** We modify the proof of Lemma 6.15 as follows. First, replace references to Lemmas 6.5, 6.6, 6.7 and 6.14 with references to Lemmas 7.5, 7.6, 7.8, and 7.10. Next, noting that the subroutines HelperBegin, HelperEnd and HelperCC do not write any of the shared variables accessed in $I_E'$, except possibly the field $V$ of a block, at line 202, we extend the case analysis as follows:

**Case F:** $O_p$ falls under Definition 6.8 (g), and $p$ writes $x \triangleright V$ in some block $x$ at line 202 during the counterpart of $O_p$ in $H$. In this case, $s_k = (s, t_k, 0)$ for some $t_k$, and $O_p$ is an ECAS operation execution.

As in Case D, it follows that $p$ reads the value $\nu_{k-1}.V$ from $x \triangleright V$, and $\nu_{k-1}.Linked[p]$ from $x \triangleright Linked[p]$. Consequently, $O_p$ returns $\langle \nu_{k-1}.V, true \rangle$ in $H$. As in Case D, it follows from the failure of the tests at line 163 and line 165 that $O_p$ is successful.
$S(i)$ (a) follows by the action of step $t_i$ by $p_i$, which does not change the current block but overwrites $x_i \triangleright V$ with new. $S(i)$ (b) holds since $Op_i$ returns $(\nu_{i-1},V,\text{true})$ in $\bar{H}$ and since $Op_i$ is a successful ECAS operation execution.

Having established linearizability (Lemmas 7.9 and 7.11), we now consider the termination and RMR complexity.

**Lemma 7.12.** The implementation $I_E'$ satisfies Specification 6.2 (termination).

**Proof.** Let $H$ be a fair history of $I_E'$, and suppose for contradiction that some operation execution $Op$ on the target object does not terminate in $H$. It follows from the structure of the access procedures of $I_E'$ that $p$ makes a non-terminating call to $\text{HelperBegin}$, $\text{HelperEnd}$ or $\text{HelperCC}$ during $Op$. It follows from the algorithms for the subroutines under consideration that one of the following cases applies:

**Case A:** $p$ loops forever at line 191 during a call to $\text{HelperBegin}(x)$. Note that by that time, $p$ has already executed line 189 during some call to $\text{HelperBegin}(x)$, and so it follows by Lemma 7.7 (a) that any subsequent call to $\text{TryToReuseBlock}(x)$ by process $w = x \triangleright \text{writer}$ returns false. Next, note that there is at most one such call by $w$ by Lemma 7.7 (c). This implies that $x \triangleright \text{changing} = \text{false}$ holds after $w$’s last call to $\text{TryToReuseBlock}(x)$ because only $w$ can assign $x \triangleright \text{changing}$ to true, namely at line 209, and following each execution of this line process $w$ resets $x \triangleright \text{changing}$ at line 215 (since $H$ is fair). But this contradicts the hypothesis of Case A, which implies that $p$ repeatedly reads $x \triangleright \text{changing} = \text{true}$ at line 191.

**Case B:** $p$ loops forever at line 217 during a call to $\text{DoneReusingBlock}(x)$ for some block $x$ such that $p = x \triangleright \text{writer}$. Then $p$ previously read $x \triangleright \text{seen} = \text{true}$ at line 216, and so by the algorithm some process $q$ began executing $\text{HelperBegin}(x)$. Since $p = x \triangleright \text{writer}$, it follows from $q \neq p$ and the algorithm (line 200) that $q$ does not subsequently execute $\text{DoneReusingBlock}(x)$ before calling $\text{HelperEnd}(x)$. In particular, $q$ does not access block $x$ at line 217. Consequently, by Case A, the fairness of $H$, and the algorithms for $\text{Read}$, $\text{LL}$ and $\text{ECAS}$, $q$ eventually completes a call to $\text{HelperEnd}(x)$ (at line 150, line 158 or line 183). Moreover, during the first such call in $H$ it assigns $x \triangleright \text{accessed} = \text{true}$ at line 196. Since no process ever writes $x \triangleright \text{accessed} = \text{false}$ by the algorithm, this contradicts the hypothesis of Case B.

Next, we analyze the RMR complexity and locality properties of the implementation $I_E'$. Recall that processes incur RMRs while accessing the block manager, block alloca-
Lemma 7.13. For any history $H$ of $I_E$, for any block $x$ accessed in $H$, and for any process $p$, $p$ incurs $O(1)$ RMRs in the CC model accessing the block manager and block allocator during operation executions on the target object where it accesses block $x$.

Proof. First, consider the block manager object $M$. By Specification 5.1, there is at most one operation execution where $p$ allocates $x$, in which case $p$ applies a successful $M.chngCurBlock(..., x)$ in the same operation execution. There is at most one other operation execution where $p$ accesses $x$ and calls $M.chngCurBlock$, namely one where $p$ calls $M.getCurBlock()$ with response $x$ and then applies a failed $M.chngCurBlock(x, ...)$, after which point $x$ is never again current (and hence is not accessed again by $p$) by Lemma 7.5. Thus, there are at most two operation executions in $H$ where $p$ accesses block $x$ and calls $M.chngCurBlock$ or $AllocBlock$, each call incurring $O(1)$ RMRs. Next, consider operation executions where $p$ calls $M.getCurBlock()$ only (and not $M.chngCurBlock$ or $AllocBlock$). In these cases $getCurBlock$ must return $x$, otherwise $p$ does not access block $x$. Process $p$ incurs $O(1)$ RMRs in $H$ accessing $M$ in such operation executions by the locality-property of $M$ (see Definition 7.2) because $M$ does not change state between $p$’s first and last $getCurBlock$ in $H$ that returns $x$ by Lemma 7.5. (Recall our assumption at the beginning of this section that $M$ satisfies the locality property for the write-through CC model. We establish this property for the block manager implementation presented in Chapter 5 later on in Lemma 7.17.)

Lemma 7.14. For any history $H$ of $I_E$, for any block $x$ accessed in $H$, and for any process $p$, the number of RMRs that $p$ incurs while accessing block $x$ in $H$ is:

- $O(1)$ in the CC model with write-back caching; and
- $O(1 + m)$ in the CC model with write-through caching, where $m$ is the number of write-like operation executions in $H$ on the target object during which $p$ accesses block $x$.

Proof. Since there are $O(1)$ fields (i.e., shared objects and subroutines) in block $x$, it suffices to show the stated upper bound on RMRs separately for each field. (In most cases, we will not distinguish between the write-through and write-back CC model, as the cost is $O(1)$ in both.)
Case A: variable $V$. Only one process can write $V$, namely the process $w$ that allocated block $x$. The ID of this process is stored in $x \triangleright writer$ once $w$ completes line 172 of ECAS following the allocation of $x$. If $p = w$, then $p$ can only write $x \triangleright V$ at line 173 (in ECAS, before block $x$ becomes current) and at line 202 (in HelperCC). On the other hand, every process may read $x \triangleright V$ at line 149, line 154 or line 162, while executing an access procedure.

Subcase A-i: $p = w$, write-back model. Process $p$ holds $x \triangleright V$ in exclusive mode in its cache from the moment it first writes it until some process $q \neq p$ reads it. Therefore, $p$ incurs $O(1)$ RMRs accessing $x \triangleright V$ until some process $q \neq p$ reads it. By the algorithm, $q$ has completed a call to HelperBegin$(x)$ by that time, and so by Lemma 7.7 (a) and the algorithm, $p$ does not write $x \triangleright V$ again. As noted earlier, this implies that no process writes $x \triangleright V$, and so subsequent accesses to $x \triangleright V$ by any process are in-cache reads that incur $O(1)$ RMRs in total per-process. Thus, the total cost of $p$ accessing $x \triangleright V$ is $O(1)$ RMRs.

Subcase A-ii: $p = w$, write-through model. Here $w$ incurs an RMR each time it writes $V$, which occurs at most once during an operation execution where $p$ allocates $x$, and then once per successful ECAS operation execution (see Case F in the proof of Lemma 7.11), which is write-like. Also, $p$ incurs $O(1)$ RMRs reading $x \triangleright V$, as in the write-back CC model (Subcase A-i). Thus, $p$ incurs $O(m)$ RMRs accessing $x \triangleright V$.

Subcase A-iii: $p \neq w$. Here $p$ only reads $x \triangleright V$. As argued in Subcase A-i, such reads occur only after the last write to $x \triangleright V$ (including initialization). Since $p$ does not write $x \triangleright V$ in this case, all accesses to $x \triangleright V$ by $p$, except the first, are in-cache. Thus, $p$ incurs $O(1)$ RMRs in total accessing $x \triangleright V$.

Case B: array $Linked[1..N]$. Note that only $p$ accesses $x \triangleright Linked[p]$, and $p$ writes this variable at most once. Thus, $p$ incurs at most two RMRs accessing this variable.

Case C: variable $writer$. This variable is written exactly once, by the process that allocates the block $x$. All other accesses are reads. Consequently, each process incurs at most two RMR accessing $writer$.

Case D: variables $seen$ and $seenBy[1..N]$. The array $x \triangleright seenBy[1..N]$ is accessed similarly to $x \triangleright Linked[1..N]$, and so the analysis is analogous. Next, consider $x \triangleright seen$, which is accessed only at line 189, line 210, and line 216. Because of the test at line 187, the assignment at line 188, and the fact that $x \triangleright seenBy[p]$ is never assigned false, $p$ accesses $x \triangleright seen$ at line 189 at most once. On the other hand, $p$ may access $x \triangleright seen$ multiple times at line 210 and line 216. However, at most once such access reads true,
Chapter 7. Locally-Accessible Implementations

since in that case \texttt{TryToReuseBlock}(x) returns \texttt{false}, and this happens at most once by Lemma 7.7 (c). Since \texttt{true} is the only value that can be written to \texttt{x \triangleright seen}, namely at line 189, it follows that all accesses to \texttt{x \triangleright seen} during a call to \texttt{TryToReuseBlock}(x) are in-cache, except the first and possibly the last (which returns \texttt{true}). Thus, \( p \) incurs \( O(1) \) RMRs in total accessing \texttt{seen} and \texttt{seenBy[1..N]}.

Case E: variables \texttt{accessed} and \texttt{accessedBy[1..N]}. The array \texttt{x \triangleright accessedBy[1..N]} is accessed similarly to \texttt{x \triangleright Linked[1..N]}, and so the analysis is analogous. Next, consider \texttt{x \triangleright accessed}, which is accessed only at line 196 and line 217. Because of the test at line 194, the assignment at line 195, and the fact that \texttt{x \triangleright accessedBy[p]} is never assigned \texttt{false}, \( p \) accesses \texttt{x \triangleright accessed} at line 196 at most once. At line 217, \( p \) reads \texttt{x \triangleright accessed} repeatedly until it reads \texttt{true}. Since \texttt{x \triangleright accessed} is only written at line 196, and the value written there is \texttt{true}, it follows that \( p \) incurs at most two RMRs at line 217. Thus, \( p \) incurs \( O(1) \) RMRs in total accessing \texttt{x \triangleright accessed} and \texttt{x \triangleright accessedBy[1..N]}.

Case F: variable \texttt{changing}. Let \( w \) denote the process that allocated block \( x \) (i.e., \texttt{x \triangleright writer}). If \( p = w \), then by the tests at line 186 and line 193, \( p \) only accesses \texttt{x \triangleright changing} at line 209 of \texttt{TryToReuseBlock}, and at line 215 of \texttt{DoneReusingBlock}. Similarly, if \( p \neq w \) then \( p \) accesses \texttt{x \triangleright changing} only at line 191 of \texttt{HelperBegin}. (This includes the case when \( x \) is the initial block and \texttt{x \triangleright writer} = \perp.)

Subcase F-i: \( p = w \), write-back model. Process \( p \) holds \texttt{x \triangleright changing} in exclusive mode in its cache from the moment it first writes it until some process \( q \neq p \) reads it at line 191. Once \( q \) reaches line 191, it follows by Lemma 7.7 (a) that \( p \) does not make another call to \texttt{TryToReuseBlock}(x), and so it executes line 209 and line 215 at most one more time with \( d = x \). Thus, \( p \) performs at most three RMRs accessing \texttt{x \triangleright changing}.

Subcase F-ii: \( p = w \), write-through model. Each write of \texttt{x \triangleright changing} at line 209 or line 215 occurs during a successful \texttt{ECAS} operation execution by \( p \) in \( H \), and incurs one RMR. Since there are at most two such writes per \texttt{ECAS} operation execution, \( p \) incurs \( O(m) \) RMRs accessing \texttt{x \triangleright changing}.

Subcase F-iii: \( p \neq w \). After \( p \) accesses \texttt{x \triangleright changing} for the first time at line 191 of \texttt{HelperBegin}, process \( w \) writes \texttt{x \triangleright changing} at most twice more, as explained in Subcase F-i. Thus, after at most three RMRs, \( p \) holds \texttt{x \triangleright changing} in its cache and can read it locally. Since \( p \) does not write \texttt{x \triangleright changing}, this implies that \( p \) incurs \( O(1) \) RMRs accessing \texttt{x \triangleright changing}.

\( \square \)

Lemma 7.15. Implementation \( I'_E \) satisfies the locality property in the write-through
and write-back CC model (see Definitions 7.2 and 7.3).

Proof. Consider any history \( H \) of \( I'_E \), and consider the linearization \( \bar{H} \) of \( H|O_\tau \) defined in our proof of conformity to type \( \tau_{ECAS} \) (see Lemma 7.11), where \( O_\tau \) is the target object. To prove the locality property, we will show that \( p \) incurs \( O(1) \) RMRs in \( H \) while executing the counterparts of certain operation executions in \( \bar{H} \). To that end, we will show that \( p \) accesses \( O(1) \) blocks during these operation executions, in which case the number of RMRs that \( p \) incurs is also \( O(1) \) by Lemma 7.13 and Lemma 7.14. (There can be at most one operation execution where \( p \) does not access any block, namely a pending one, and it follows easily that \( p \) incurs \( O(1) \) in that operation execution as well.)

**Property (R)** (Definition 7.2). We must consider the write-through and write-back CC models. Fix process \( p \) and a sequence \( \bar{H}' \) of consecutive read-like operation executions in \( \bar{H} \). Let \( H' \) denote the sequence of atomic steps (which access base objects) in \( H \) corresponding to \( \bar{H}' \). It suffices to show that \( p \) accesses at most three blocks in \( H' \).

First, we will show that \( p \) allocates at most one block in \( H' \). Suppose, for contradiction, that \( p \) allocates two or more blocks. Then this happens during the counterparts of at least two distinct ECAS operation executions by \( p \) in \( \bar{H}' \), say \( Op_1 \) and \( Op_2 \) (in that order), whose timestamps fall under Definition 6.8 (d) or (e). In fact, clause (e) must apply to both because, by our analysis in the proof of Lemma 6.15, operation executions of the other type are write-like. Consequently, by the Definition 6.8 and Definition 6.9, there is an ECAS operation execution \( Op_e \) whose timestamp falls under Definition 6.8 (d), and which is linearized between \( Op_1 \) and \( Op_2 \) in \( \bar{H} \), hence in \( \bar{H}' \). Since \( Op_e \) is write-like by our analysis in the proof of Lemma 6.15, this contradicts the definition of \( \bar{H}' \).

Second, we will show that \( M.getCurBlock() \) returns at most two distinct values to \( p \) during \( H' \). Suppose, for contradiction, that \( p \) receives three such values, say during the counterparts of operation executions \( Op_1, Op_2, \) and \( Op_3 \) (in that order) in \( \bar{H}' \). Then by Definition 6.8 and Definition 6.9, there is an ECAS operation execution \( Op_e \) in \( \bar{H} \) whose counterpart in \( H \) changes the current block. Furthermore, its timestamp falls under Definition 6.8 (d), and it is linearized between \( Op_1 \) and \( Op_2 \) in \( \bar{H} \), hence in \( \bar{H}' \). (Process \( p \) may not “see” the block made current by \( Op_e \) until \( Op_3 \), which is why we consider three operation executions by \( p \).) Again, this contradicts the definition of \( \bar{H}' \).

Thus, \( p \) allocates at most one block and receives the addresses of at most two more from \( M.getCurBlock \) in \( H' \), and so \( p \) accesses at most three blocks in \( H' \), as wanted.

**Property (W)** (Definition 7.3). We need only consider the write-back CC model. Fix
process $p$ and a sequence $\hat{H}'$ of consecutive operation executions by $p$ in $\hat{H}$. Again let $H'$ be the corresponding sequence of atomic steps. It suffices to show that $p$ accesses at most four blocks in $H'$.

As in the proof of (R), note that if $M$.getCurBlock() returns three distinct values to $p$ during $H'$, then there is an ECAS operation execution $Op_e$ in $\hat{H}'$ whose counterpart in $H$ changes the current block. In this case, this does not lead to a contradiction, since $\hat{H}'$ may contain write-like operation executions, but it tells us that $p$ executes $Op_e$. Furthermore, we can generalize the argument easily and conclude that if $M$.getCurBlock() returns $k$ distinct values to $p$, then $p$ performs at least $\min(0, k - 2)$ ECAS operation executions in $H'$ whose counterparts in $H$ change the current block. Call this observation (⋆).

Next, we will show that $p$ tries to change the current block at most once in $H'$. Suppose, for contradiction, that $p$ does so twice, say during the counterparts of operation executions $Op_1$ and $Op_2$ (in that order) in $\hat{H}'$. Note that the timestamps of $Op_1$ and $Op_2$ fall under Definition 6.8 (d) or (e). Suppose that $p$’s $M$.getCurBlock() during the counterpart of $Op_2$ in $H$ returns $x$. It follows from the algorithm and Definition 6.8 (d) that $p$ completes a call to $\text{TryToReuseBlock}(x)$ during the counterpart of $Op_2$ in $H$, with response false, and so Lemma 7.7 (b) implies two things. First, some process $q \neq p$ has made a call to $\text{HelperBegin}(x)$ by that point in $H$. Second, since $p$ completes its subsequent call to $\text{DoneReusingBlock}(x)$ during the counterpart of $Op_2$ in $H$ (otherwise it cannot call $M$.chngCurBlock), some process $r$ has made a call to $\text{HelperEnd}(x)$ (possibly $r = q$ but not necessarily). It follows from the latter point, Definition 6.8, and Definition 6.9 that $r$’s operation execution, say $Op_r$, which also accesses $x$, appears in $\hat{H}$ and is linearized before $Op_2$.

We will now show that $Op_r$ is linearized after $Op_1$, which contradicts $\hat{H}'$ containing operation executions by $p$ only (since $Op_r$ is linearized before $Op_2$), and implies that $p$ changes the current block at most once in $H'$. Suppose otherwise, and recall that $Op_1$’s timestamp falls under Definition 6.8 (d) or (e). Assuming, without loss of generality, that $Op_1$ and $Op_2$ are the first two operation executions in $\hat{H}'$ whose counterparts in $H$ change the current block, it follows that $Op_1$’s timestamp is $(x, t_1, ...)$ for some $t_1$. (Recall that $Op_1$ and $Op_2$ fall under Definition 6.8 (d), that $p$ calls $M$.getCurBlock() with response $x$ during the counterpart of $Op_2$ in $H$, and that no process other than $p$ applies an operation execution between $Op_1$ and $Op_2$ in $\hat{H}$.) At the same time, since $r$’s call to $M$.getCurBlock() during the counterpart of $Op_r$ in $H$ returns $x$, by Definition 6.8 $Op_r$’s timestamp is of the form $(x', t_r, ...)$ for some $t_r$, where either $x' = x$ or $x'$ is a block
that becomes current after \( x \). Furthermore, if \( x' = x \), then \( t_1 < t_r \) by Lemma 7.5 and because step \( t_1 \) in \( H \) (which could be by \( p \) or another process) makes \( x \) current, whereas \( r \) makes a call to \( M \).getCurblock() with response \( x \) before step \( t_r \) in \( H \). In either case, \( Op_r \) is linearized after \( Op_1 \) by Definition 6.9, as wanted.

Thus, \( p \) tries to change the current block at most once in \( H' \). Consequently, by our earlier observation (⋆), \( M \).getCurblock() returns at most three distinct values to \( p \) in \( H' \). Since \( p \) allocates a block only in operation executions where it tries to change the current block (by calling \( M \).chgCurblock), and it does so at most once per operation execution, \( p \) accesses at most four blocks in total in \( H' \), as wanted.

\[ \square \]

**Theorem 7.16.** The implementation \( I'_E \) satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs \( O(1) \) RMRs. Finally, \( I'_E \) satisfies the locality property in the write-through and write-back CC models (Definitions 7.2 and 7.3).

**Proof.** Specification 6.1 under Condition 6.4 follows directly from Lemma 7.9 and Lemma 7.11. Specification 6.2 follows directly from Lemma 7.12. \( O(1) \) RMR complexity follows from Lemma 7.13, Lemma 7.14, and the fact that a process accesses at most two blocks during any operation execution on the target object. (It follows easily that an operation execution where no block is accessed also incurs \( O(1) \) RMRs.) The locality property follows from Lemma 7.15.

To complete our analysis in this section, we prove that the block manager implementation from Chapter 5 satisfies the locality property necessary for our purposes in this chapter. As stated earlier, we require only that the block manager satisfy the locality property for the write-through CC model (see Definition 7.2), even when used in the write-back CC model. (If the block manager satisfied the locality property in the write-back CC model, this would not help because every operation execution on the ECAS object that applies a write-like operation on the block manager already incurs RMRs for another reason – it calls AllocBlock.) To define this locality property, we classify getCurblock operations as read-like, failed chgCurblock operations as read-like and successful chgCurblock operations as write-like.

**Lemma 7.17.** The block manager implementation \( I_{BM} \) from Chapter 5 satisfies the locality property stated in Definition 7.2 in both the write-through and write-back CC
models provided that whenever a process $p$ calls $\text{chngCurBlock}(x, y)$, $p$’s last operation execution on the block manager was a $\text{getCurBlock()}$ that returned $x$.

Proof. Consider a history $H$ of the implementation $I_{BM}$ where calls to $\text{chngCurBlock}$ are restricted, as stated. Let $\bar{H}$ be the linearization of $H|O_\tau$ established in Chapter 5, where $O_\tau$ denotes the target object. We must prove property (R), stated in Definition 7.2. Fix $p$ and $\bar{H}'$ as in Definition 7.2, and consider the sequence $H'$ of base object atomic steps corresponding to $\bar{H}'$ in $H$. We must show that $p$ incurs $O(1)$ RMRs in $H$ executing the atomic steps in $H'$. Note that no process writes $D$ in $H'$ because at that point a successful $\text{chngCurBlock}$ takes effect, which is write-like, and yet we assume that $\bar{H}'$ does not contain any write-like operation executions. Consequently, each access to $D$ by $p$ in $H'$ is a read, and at most one causes an RMR. Furthermore, each such read returns the same value, say $x$, and so by the algorithm, $x \triangleright winner$ is not written after $p$’s first read of $D$ in $H'$, which implies that $p$ incurs at most one RMR accessing $x \triangleright winner$. Finally, consider pseudo-locks. It follows from the algorithm (see Figure 4.1) and since $p$ only reads $x$ from $D$ that the only pseudo-lock $p$ accesses in $H'$ is the one in block $x$. Process $p$ calls $x \triangleright \text{Pseudo-Enter()}$ and $x \triangleright \text{Pseudo-Exit()}$ at most once by Lemma 5.5 (a), and each call incurs $O(1)$ RMRs by Theorem 4.5. \qed
7.3 Locally-Accessible ECAS for the DSM Model

Recall that the locality property in the DSM model states that for some designated process \( p_{\text{special}} \), each operation execution on the target object applied by \( p_{\text{special}} \) should cost zero RMRs. The implementation \( I_E \) presented in Chapter 6 does not satisfy this because \( p_{\text{special}} \) may perform RMRs while accessing the block manager. This is problematic because our general approach for implementing the block manager (in Chapter 5) is, informally speaking, incompatible with the DSM locality property: We rely on the ability of a process executing a failed \texttt{chgCurBlock} operation execution to wait for a concurrent successful \texttt{chgCurBlock} operation execution to take effect, so that the former can be linearized after the latter. (This aspect of the implementation is handled by the pseudo-lock.) If process \( p_{\text{special}} \) is the one being waited for, it cannot signal the waiting processes without performing at least one RMR if those waiting processes have all entered local-spin busy-wait loops.

In the remainder of this section we show how to modify the implementation \( I_E \) of ECAS from Chapter 6 to achieve the locality property in the DSM model with respect to a designated process \( p_{\text{special}} \). To that end, we construct an ECAS implementation \( I_{E-\text{DSM}} \) by taking \( I_E \) and making the base objects used therein locally-accessible to \( p_{\text{special}} \). (Note that we revert to the trivial implementations of the subroutines \texttt{HelperBegin}, \texttt{HelperEnd}, and \texttt{HelperCC}, as shown in Figure 6.7.)

Recall that the base objects used in \( I_E \) are the block manager \( M \), and for each block the following: registers \( V \) and \( \text{writer} \), and the arrays \( \text{NextVal}[1..N] \) and \( \text{Linked}[1..N] \). (We discuss the block allocator separately in Chapter 11.) We construct \( I_{E-\text{DSM}} \) from \( I_E \) by making all of these base objects local to \( p_{\text{special}} \). Note that in the case of per-block base objects, such as \( V \), this applies to \textit{all} blocks, not just ones returned by the block allocator of \( p_{\text{special}} \). This is because \( p_{\text{special}} \) may access any block, even if it that block was allocated by another process. Similarly, we must make \textit{all} elements of array \( \text{NextVal}[1..N] \), and also \( \text{Linked}[p_{\text{special}}] \), local to \( p_{\text{special}} \).

Our task of making base objects local to \( p_{\text{special}} \) is trivial for register objects, which leaves only the block manager object \( M \). The implementation \( I_{BM} \) described in Chapter 5 cannot be made local to \( p_{\text{special}} \) easily for reasons described earlier. Instead, we present a new implementation that provides separate execution paths for \( p_{\text{special}} \) and other processes, and is locally-accessible to \( p_{\text{special}} \) in the DSM model, with RMR complexity \( O(1) \) for other processes. We refer to this implementation as \( I_{BM-\text{DSM}} = (\tau_{BM}, P, B, H) \), and
to the “other processes” as non-special.

The implementation $I_{BM}$-DSM is presented in Figures 7.2–7.3. This implementation is somewhat similar to $I_{BM}$ from Chapter 5. The address of the current block is recorded using registers $D_{\text{special}}$ and $D_{\text{other}}$. (These replace the single register $D$ in $I_{BM}$.) Each register actually stores a tuple of the form $(x, s)$, where $x$ is a block address and $s$ is an integer sequence number. The register containing the higher sequence number holds the address of the current block. (In case the sequence numbers are equal, $D_{\text{other}}$ determines the current block.) The access procedure for $\text{getCurBlock()}$ reads $D_{\text{special}}$ and $D_{\text{other}}$ at lines 219–220 (using a non-atomic pair of reads), and then at lines 221–225 compares the sequence numbers, and returns the block address corresponding to the higher number.

To apply $\text{chngCurBlock}(x, y)$, processes synchronize as follows: Non-special processes compete with each other by trying to acquire a pseudo-lock in block $x$ (line 243). The winner of this pseudo-lock synchronizes with $p_{\text{special}}$ using a leader election algorithm (line 232 and line 244) that is local to $p_{\text{special}}$ (see Specification 3.9). The process $w$ that wins the LE algorithm is the one whose $\text{chngCurBlock}(x, y)$ succeeds. If $w = p_{\text{special}}$, then $w$ writes $y$ and a new sequence number $\text{nextSeq}$ to $D_{\text{special}}$ (line 233). The new sequence number is computed at lines 226–230, and the algorithm for this ensures that $\text{nextSeq}$ is higher than the sequence number in $D_{\text{other}}$. This computation uses values stored in private variables $S_{\text{special}}$ and $S_{\text{other}}$, which are assigned by an earlier call to $\text{getCurBlock()}$ (see Condition 7.18). If $w \neq p_{\text{special}}$, then $w$ follows similar steps; it computes $\text{nextSeq}$ at lines 245–249, and then writes $D_{\text{other}}$ (line 250). The sequence number chosen in that case is at least as high as the one in $D_{\text{special}}$.

When a process $p$ applies a failed $\text{chngCurBlock}(x, y)$ (i.e., it does not win LE), it must ensure that the successful $\text{chngCurBlock}(x, ...)$ by some other process $w$ has taken effect before terminating. We do this using three techniques. First, if $p \neq p_{\text{special}}$ and $w \neq p_{\text{special}}$, then $p$ waits for $w$ using a pseudo-lock (lines 243 and 261), as in $I_{BM}$. Second, if $p = p_{\text{special}}$ and $w \neq p_{\text{special}}$, $p$ waits for $w$ using a simple busy-wait loop (lines 240 and 251). Third, if $p \neq p_{\text{special}}$ and $w = p_{\text{special}}$, then $p$ cannot wait for $p_{\text{special}}$ because $p_{\text{special}}$ cannot signal $p$ later on without performing an RMR (as explained earlier); instead, $p$ applies a “helping mechanism” to ensure that $p_{\text{special}}$’s operation execution has taken effect. Here $p$ discovers what value $p_{\text{special}}$ is trying to write to $D_{\text{special}}$ (lines 231 and 255), and then $p$ writes this value to $D_{\text{special}}$ (line 256). At that point, $p_{\text{special}}$’s operation execution takes effect, unless $p_{\text{special}}$ completed its own write to $D_{\text{special}}$ at line 233 earlier. To ensure that the helping mechanism does not corrupt $D_{\text{special}}, p_{\text{special}}
waits for this “round” of the mechanism to finish before applying another \texttt{chngCurBlock} (lines \texttt{236–238, 253}, and \texttt{259}).

### Declarations

**Shared variables:** (global)

\(D_{\text{special}}, D_{\text{other}}\) – registers, store a tuple \((b, s)\) where \(b\) is a block address and \(s\) is an integer, initially \((b_0, 0)\) where \(b_0\) is the initial block, both local to \(p_{\text{special}}\)

**Shared variables:** (per-block)

\(\text{winner}\) – register, stores a process ID or \(\perp\), initially \(\perp\), local to \(p_{\text{special}}\)

\(\text{helping, specialDone, helperDone}\) – Boolean flags, initially \(\text{false}\), local to \(p_{\text{special}}\)

\(A\) – register, same type as \(D_{\text{special}}\) and \(D_{\text{other}}\), local to \(p_{\text{special}}\)

**Subroutines:** (one instance per-block)

\(\text{LeaderElect}() – O(1)\) leader election algorithm local to \(p_{\text{special}}\) (see Chapter 3.3)

\(\text{Pseudo-Enter()/Pseudo-Exit()} – O(1)\)-RMR pseudo-lock from Chapter 4

**Private variables:** (per-process)

\(B_{\text{special}}, B_{\text{other}}\) – block addresses, uninitialized

\(S_{\text{special}}, S_{\text{other}}\) – integers, uninitialized

\(B\) – same type of value as \(D_{\text{special}}\) and \(D_{\text{other}}\)

---

**Figure 7.2:** Block manager implementation for DSM model – part 1 (declarations).

Before we begin our analysis, we introduce some useful notation for this chapter. When referring to the state of \(D_{\text{special}}\) and \(D_{\text{other}},\) or a value read from these registers, we denote by operators \texttt{Block()} and \texttt{Seq()} the block address and sequence number embedded therein. For any history \(H\) of \(I_{BM-DSM},\) and any state \(H[i]\) that occurs in \(H,\) we define \(\text{MaxBlock}(H[i])\) as \(\text{Block}(D_{\text{special}})\) if \(\text{Seq}(D_{\text{special}}) > \text{Seq}(D_{\text{other}})\) in state \(H[i]\), and as \(\text{Block}(D_{\text{other}})\) otherwise. We also define \(\text{MaxSeq}(H[i])\) as \(\max(\text{Seq}(D_{\text{special}}), \text{Seq}(D_{\text{other}}))\) in state \(H[i].\)

The correctness of the implementation \(I_{BM-DSM}\) is contingent on the following simplifying condition, which we justify later (see Lemma 7.34):

**Condition 7.18.** If a process \(p\) calls \texttt{chngCurBlock}(\(x, y\)), then:

(a) \(p\)'s last operation execution on the target object was a \texttt{getCurBlock} that returned \(x;\) and

(b) \(p\) never before invoked \texttt{chngCurBlock}(\(x, ...\)); and

(c) no process has invoked \texttt{chngCurBlock}(\(..., y\)) and \(y\) is not the initial block.
Function `getCurBlock()`

219 \((B_{special}, S_{special}) := \text{read}(D_{special})\)

220 \((B_{other}, S_{other}) := \text{read}(D_{other})\)

221 if \(S_{other} < S_{special}\) then

222 \quad \text{return } B_{special}

223 else

224 \quad \text{return } B_{other}

225 end

Function `chngCurBlock(x, y)` for non-special processes

226 if \(S_{other} < S_{special}\) then

227 \quad nextSeq := S_{special}

228 else

229 \quad nextSeq := S_{other} + 1

230 end

231 write \(x \triangleright A := (y, nextSeq)\)

232 if \(x \triangleright \text{LeaderElect()} = \text{win}\) then

233 \quad write \(D_{special} := (y, nextSeq)\)

234 \quad write \(x \triangleright \text{winner} := p_{special}\)

235 \quad write \(x \triangleright \text{specialDone} := \text{true}\)

236 if read\((x \triangleright \text{helping}) = \text{true}\) then

237 \quad \text{await } x \triangleright \text{helperDone} = \text{true}

238 \quad \text{end}

239 else

240 \quad \text{await } x \triangleright \text{winner} \neq \bot

241 \quad \text{end}

242 return \text{read}(x \triangleright \text{winner})

Function `chngCurBlock(x, y)` for non-special processes

243 if \(x \triangleright \text{Pseudo-Enter()} = \text{true}\) then

244 \quad if \(x \triangleright \text{LeaderElect()} = \text{win}\) then

245 \quad \quad if \(S_{other} < S_{special}\) then

246 \quad \quad \quad nextSeq := S_{special}

247 \quad \quad else

248 \quad \quad \quad nextSeq := S_{other}

249 \quad \quad end

250 \quad \quad write \(D_{other} := (y, nextSeq)\)

251 \quad \quad write \(x \triangleright \text{winner} := \text{PID}\)

252 \quad \quad \text{else}

253 \quad \quad \quad write \(x \triangleright \text{helping} := \text{true}\)

254 \quad \quad \quad \text{if read}(x \triangleright \text{specialDone}) = \text{false} then

255 \quad \quad \quad \quad \quad \quad B := \text{read}(x \triangleright A)

256 \quad \quad \quad \quad \quad \quad write \(D_{special} := B\)

257 \quad \quad \quad \quad \quad \quad write \(x \triangleright \text{winner} := p_{special}\)

258 \quad \quad \quad \text{end}

259 \quad \quad \quad write \(x \triangleright \text{helperDone} := \text{true}\)

260 \quad \quad \quad \text{end}

261 \quad \quad \quad \text{end}

262 \quad \quad \text{end}

263 \quad \text{return } \text{read}(x \triangleright \text{winner})

Figure 7.3: Block manager implementation for DSM model – part 2 (access procedures).
We now establish several lemmas that are used later on for proving linearizability.

**Lemma 7.19.** For any history \( H \) of implementation \( I_{BM-DSM} \) and for any block \( x \) accessed by any process in \( H \), the leader election algorithm and pseudo-lock in block \( x \) are accessed according to Conditions 3.4 and 4.1.

**Proof.** This follows immediately from Condition 7.18 (b) and the structure of the access procedures.

**Lemma 7.20.** For any history \( H \) of implementation \( I_{BM-DSM} \), and any block \( x \), let \( H' \) be the subsequence of atomic steps in \( H \) applied in executions of \( \text{chngCurBlock}(x, ...) \). Then \( H' \) either contains at most two writes to \( D_{special} \) (i.e., at most one by \( p_{special} \) and at most one by a non-special process) and zero writes to \( D_{other} \), or it contains zero writes to \( D_{special} \) and at most one write to \( D_{other} \) (which is by a non-special process).

**Proof.** A write to \( D_{special} \) can only occur at line 233 or line 256, and a write to \( D_{other} \) can only occur at line 250. By Condition 7.18 (b) and the algorithm, \( H' \) contains at most one write to \( D_{special} \) or \( D_{other} \) by \( p_{special} \), and this is a write to \( D_{special} \). Similarly, for any non-special process, \( H' \) contains at most one write to \( D_{special} \) or \( D_{other} \). Furthermore, any non-special process that applies such a write must first acquire the pseudo-lock in block \( x \) at line 243, and so there is at most one such process by Lemma 7.19 and Specification 4.2 (b). Thus, \( H' \) contains at most one write to \( D_{special} \) or \( D_{other} \) by any non-special process.

To complete the proof, it suffices to rule out the case when \( H' \) contains writes to both \( D_{special} \) and \( D_{other} \). As explained earlier, it follows in this case that a non-special process \( q \) writes \( D_{other} \) and that \( p_{special} \) writes \( D_{special} \). Furthermore, by the algorithm \( p_{special} \) wins \( x \triangleright \text{LeaderElect}() \) at line 232 before applying its write at line 233, and \( q \) wins \( x \triangleright \text{LeaderElect}() \) at line 244 before applying its write at line 250. Thus, \( p_{special} \) and a non-special process both win \( x \triangleright \text{LeaderElect}() \) in \( H \), which contradicts Lemma 7.19 and Specification 3.5.

**Lemma 7.21.** For any history \( H \) of implementation \( I_{BM-DSM} \), and any block \( x \), the variable \( x \triangleright \text{winner} \) changes state at most once in \( H \).

**Proof.** Two writes to \( x \triangleright \text{winner} \) that assign different values can occur only at line 234 and line 251, or at line 251 and line 257. In the first case, a write to \( D_{special} \) (at line 233) occurs during some \( \text{chngCurBlock}(x, ...) \) operation execution, and a write to
$D_{other}$ (at line 250) also occurs during some $\text{chngCurBlock}(x,...)$, which contradicts Lemma 7.20. In the second case, a write to $D_{other}$ (at line 250) occurs during some $\text{chngCurBlock}(x,...)$ operation execution, and a write to $D_{special}$ (at line 256) also occurs during some $\text{chngCurBlock}(x,...)$, which again contradicts Lemma 7.20.

Lemma 7.22. For any history $H$ of implementation $I_{BM-DSM}$, any process $q \neq p_{special}$, and any block $x$, whenever $q$ is at lines 255–256 during a $\text{chngCurBlock}(x,...)$ operation execution, $p_{special}$ is continuously in a pending $\text{chngCurBlock}(x,...)$ operation execution where it has completed line 231 and not yet completed line 238.

Proof. Let $Op_{q}$ denote $q$’s $\text{chngCurBlock}(x,...)$ operation execution under consideration and note that by Condition 7.18 (b) there is at most one such operation execution in $H$.

First, we will prove that when $q$ reaches line 255 during $Op_{q}$, process $p_{special}$ has already completed line 231 during its own $\text{chngCurBlock}(x,...)$ operation execution. Note that before reaching line 255, $q$ acquires the pseudo-lock in block $x$ at line 243, and then loses $x \triangleright \text{LeaderElect}(\cdot)$ at line 244. Since $q$ loses $x \triangleright \text{LeaderElect}(\cdot)$, by Specification 3.5 another process must make a call to $x \triangleright \text{LeaderElect}(\cdot)$ before $q$ completes its call. By Lemma 7.19, Specification 4.2 (b) and the algorithm, the only other process that may call $x \triangleright \text{LeaderElect}(\cdot)$ is $p_{special}$ at line 232 during a $\text{chngCurBlock}(x,...)$ operation execution. Now let $Op_{special}$ denote this $\text{chngCurBlock}(x,...)$ operation execution by $p_{special}$ and note that it is unique in $H$ by Condition 7.18 (b). Since $p_{special}$ makes its call to $x \triangleright \text{LeaderElect}(\cdot)$ at line 232 before $q$ completes its own call at line 244, $p_{special}$ completes line 231 during $Op_{special}$ before $q$ reaches line 255 during $Op_{q}$, as wanted.

To complete the proof, we will show that $q$ completes line 256 during $Op_{q}$ before $p_{special}$ completes line 238 during $Op_{special}$. If $q$ reaches line 256 at all during $Op_{q}$, it does so after reading $x \triangleright \text{specialDone} = \text{false}$ at line 254, which by the algorithm occurs before $p_{special}$ writes $x \triangleright \text{specialDone} = \text{true}$ at line 235. Thus, $p_{special}$ reads $x \triangleright \text{helping} = \text{true}$ at line 236 after $q$ writes $x \triangleright \text{helping} = \text{true}$ at line 253, and so $p_{special}$ branches to line 237 during $Op_{special}$. Before $p_{special}$ completes this line, some process $z$ writes $x \triangleright \text{helperDone} = \text{true}$, at line 259 during a $\text{chngCurBlock}(x,...)$ after acquiring the pseudo-lock in block $x$ at line 243. Since $q$ wins this pseudo-lock before reaching line 238, it follows from Lemma 7.19 and Specification 4.2 (b) that $z = q$. Since $Op_{q}$ is $q$’s only $\text{chngCurBlock}(x,...)$ in $H$, as explained earlier, this implies that $q$ completes line 259 during $Op_{q}$ before $p_{special}$ completes line 237 during $Op_{special}$; hence $q$ completes line 256 during $Op_{q}$ before $p_{special}$ completes line 238 during $Op_{special}$, as wanted.
Lemma 7.23. For any history $H$ of implementation $I_{BM-DSM}$, and for any block $x$, if multiple $\text{chngCurBlock}(x, ...)$ operation executions occur in $H$, then the earliest write to $D_{\text{special}}$ or $D_{\text{other}}$ that occurs during these is the only such write that changes the state of $D_{\text{special}}$ or $D_{\text{other}}$.

Proof. If multiple writes to $D_{\text{special}}$ or $D_{\text{other}}$ occur in $\text{chngCurBlock}(x, ...)$ operation executions, then by Lemma 7.20 there are exactly two of these, one by $p_{\text{special}}$ and one by some non-special process $q$. Furthermore, both of these writes apply to $D_{\text{special}}$.

To prove the lemma, we first show that $q$’s write at line 256 assigns the same value as the write by $p_{\text{special}}$ at line 233 during the two $\text{chngCurBlock}(x, ...)$ operation executions under consideration. To that end, note that by Lemma 7.22, $q$’s read of $x \triangleright A$ at line 255 occurs after $p_{\text{special}}$’s write of $x \triangleright A$ at line 231 during these operation executions. Since $x \triangleright A$ is written at most once in $H$ by the algorithm and Condition 7.18 (b), the value $p_{\text{special}}$ writes to $x \triangleright A$ at line 231 is the same as the value $q$ reads from $x \triangleright A$ at line 255, hence the same as the value $q$ writes to $D_{\text{special}}$ at line 256.

Finally, we must prove that no process $z$ writes $D_{\text{special}}$ between the two writes under consideration by $p_{\text{special}}$ and $q$, regardless of the order in which they occur. First, note that by Lemma 7.22 and the algorithm, $p_{\text{special}}$ is continuously in a $\text{chngCurBlock}(x, ...)$ operation execution between these two writes. Since $p_{\text{special}}$ writes $D_{\text{special}}$ at most once in each $\text{chngCurBlock}$, this implies $z \neq p_{\text{special}}$. Furthermore, if $z \neq p_{\text{special}}$ then by Lemma 7.22 $z$’s write to $D_{\text{special}}$ must occur during a $\text{chngCurBlock}(x, ...)$ by $z$, which implies that $D_{\text{special}}$ is written three times (i.e., by $p_{\text{special}}$, $q$ and $z$) in a $\text{chngCurBlock}(x, ...)$, contradicting Lemma 7.20.

Lemma 7.24. For any history $H$ of implementation $I_{BM-DSM}$, any process $p$, and any step $i$ in $H$, if $p$ writes $D_{\text{special}}$ or $D_{\text{other}}$ in step $i$ of $H$, and changes the state of the variable written to $C$, then letting $y = \text{Block}(C)$ and $x = \text{MaxBlock}(H[i - 1])$:

(a) $y = \text{MaxBlock}(H[i])$; and
(b) the sequence number in the variable written does not decrease; and
(c) $|\text{Seq}(D_{\text{special}}) - \text{Seq}(D_{\text{other}})| \leq 1$ in state $H[i]$; and
(d) the value $y$ has never been written to $D_{\text{special}}$ or $D_{\text{other}}$ before in $H$.

Proof. Since the properties stated are safety properties, it suffices to consider finite $H$. Let $S(j)$ denote parts (a)–(d) for a history $H$ of length $j$. We proceed by induction on $j$. For $S(0)$, all parts follow trivially. Now for any $j > 0$, suppose that $S(j - 1)$ holds,
and consider $S(j)$. It suffices to consider the case when $p$ writes $D_{\text{special}}$ or $D_{\text{other}}$ in step $j$ of $H$, and changes the state of the variable written to $C$. It follows from the algorithm that such a step must occur during some $\text{chgCurBlock}(x', y')$ operation execution $Op$ by $p$, for some $x'$ and $y'$.

Parts (a)–(c): Suppose that $x$ is written to $D_{\text{special}}$ or $D_{\text{other}}$ for the first time in step $j'$ (where $j' = 0$ if $x$ is the initial block $b_0$). It follows from $S(j - 1)$ (a) and the initialization of $D_{\text{special}}$ and $D_{\text{other}}$ that $x = \text{MaxBlock}(H[j'])$. Since $x = \text{MaxBlock}(H[j - 1])$ it follows from $S(j - 1)$ (a) and $S(j - 1)$ (d) that neither $D_{\text{special}}$ nor $D_{\text{other}}$ changes state between $H[j']$ and $H[j]$. Call this observation $(\star)$.

Now consider the sequence number $\text{Seq}(C)$. This number is computed and recorded in private variable $\text{nextSeq}$, either at lines 226–230 or lines 245–249, using as inputs the values $S_{\text{special}}$ and $S_{\text{other}}$ recorded at lines 219–220 during a prior call to $\text{getCurBlock}$ that returns $x$. The latter call is either by $p$ (see Condition 7.18 (b)), or by $p_{\text{special}}$ if $p$ obtains the new sequence number from $x \triangleright A$ at line 255 (see Lemma 7.22 and lines 226–231). In particular, during the computation of $\text{nextSeq}$, $S_{\text{special}}$ and $S_{\text{other}}$ are the sequence numbers read from $D_{\text{special}}$ and $D_{\text{other}}$ at lines 219–220, respectively, during a $\text{getCurBlock}$ that returns $x$. Now by our choice of $j'$, $x$ was read from either $D_{\text{special}}$ or $D_{\text{other}}$ after step $j'$, and so by $S(j - 1)$ (b) and $x = \text{MaxBlock}(H[j'])$ it follows that $\max(S_{\text{special}}, S_{\text{other}}) \geq \text{MaxSeq}(H[j'])$. At the same time, by $S(j - 1)$ (b) and observation $(\star)$, it follows that $\max(S_{\text{special}}, S_{\text{other}}) \leq \text{MaxSeq}(H[j'])$. Thus, $\max(S_{\text{special}}, S_{\text{other}}) = \text{MaxSeq}(H[j'])$, and so by the computation of $\text{Seq}(C)$ at lines 226–230 or lines 245–249, either $\text{Seq}(C) = \text{MaxSeq}(H[j'])$ or $\text{Seq}(C) = \text{MaxSeq}(H[j']) + 1$. $S(j)$ (b) follows from this and from observation $(\star)$. It remains to prove $S(j)$ (a) and $S(j)$ (c).

Case A: $p = p_{\text{special}}$ and $p$’s write in step $j$ occurs at line 233. Consider $p$’s prior computation of $\text{nextSeq}$ at lines 226–230.

If $S_{\text{other}} < S_{\text{special}}$ then $p$ read $x$ from $D_{\text{special}}$ during its last $\text{getCurBlock}$, and $\text{nextSeq} = S_{\text{special}}$. By observation $(\star)$ and our definition of $j'$, $D_{\text{special}}$ still contains $(x, S_{\text{special}})$ in state $H[j - 1]$, $D_{\text{other}}$ does not contain $(x, \ldots)$ in state $H[j - 1]$, and $x = \text{MaxBlock}(H[j - 1])$. Thus, when $p$ writes $(y, S_{\text{special}})$ to $D_{\text{special}}$ in step $j$, $\text{MaxSeq}(H[j]) = \text{nextSeq}$ and $y = \text{MaxBlock}(H[j])$, which implies $S(j)$ (a). Since $\text{Seq}(D_{\text{special}})$ does not change, $S(j)$ (c) follows from $S(j - 1)$ (c).

If $S_{\text{other}} \geq S_{\text{special}}$ then $p$ read $x$ from $D_{\text{other}}$ during its last $\text{getCurBlock}$, and $\text{nextSeq} = S_{\text{other}} + 1$. By observation $(\star)$ and our definition of $j'$, $D_{\text{other}}$ still contains $(x, S_{\text{other}})$ in state $H[j]$, $D_{\text{special}}$ does not contain $(x, \ldots)$ in state $H[j - 1]$, and
Chapter 7. Locally-Accessible Implementations

94

approach as in Chapter 5. For each history $H$, nextSeq = $\text{nextSeq}$ and $y = \text{MaxBlock}(H[j])$, which implies $S(j)$ (a). Since $\text{Seq}(D_{\text{special}}) = \text{Seq}(D_{\text{other}}) + 1$ in state $H[j]$, $S(j)$ (c) also follows.

Case B: $p \neq p_{\text{special}}$ and $p$’s write in step $j$ occurs at line 256 of $\text{chngCurBlock}$. As explained earlier, the value $p$ writes is based on the computation of nextSeq by $p_{\text{special}}$ at lines 226–230. Thus, when $p$ writes $D_{\text{special}}$ in step $j$, $S(j)$ (a) and $S(j)$ (c) follow as in Case A.

Case C: $p \neq p_{\text{special}}$ and $p$’s write in step $j$ occurs at line 250. As in Case A, consider $p$’s prior computation of nextSeq at lines 245–249.

If $S_{\text{other}} < S_{\text{special}}$ then $p$ read $x$ from $D_{\text{special}}$ during its last getCurBlock, and nextSeq = $S_{\text{special}}$. By observation (*) and our definition of $j'$, $D_{\text{special}}$ still contains $(x, S_{\text{special}})$ in state $H[j]$, $D_{\text{other}}$ does not contain $(x, ...)$ in state $H[j - 1]$, and $x = \text{MaxBlock}(H[j - 1]))$. Thus, when $p$ writes $(y, S_{\text{special}})$ to $D_{\text{other}}$ in step $j$, $\text{MaxSeq}(H[j]) = \text{nextSeq}$ and $y = \text{MaxBlock}(H[j])$, which implies $S(j)$ (a). Since $\text{Seq}(D_{\text{special}}) = \text{Seq}(D_{\text{other}})$ in state $H[j]$, $S(j)$ (c) also follows.

If $S_{\text{other}} \geq S_{\text{special}}$ then $p$ read $x$ from $D_{\text{other}}$ during its last getCurBlock, and nextSeq = $S_{\text{other}}$. By observation (*) and our definition of $j'$, $D_{\text{other}}$ still contains $(x, S_{\text{other}})$ in state $H[j]$, $D_{\text{special}}$ does not contain $(x, ...)$ in state $H[j - 1]$, and $x = \text{MaxBlock}(H[j - 1]))$. Thus, when $p$ writes $(y, S_{\text{other}})$ to $D_{\text{other}}$ in step $j$, $\text{MaxSeq}(H[j]) = \text{nextSeq}$ and $y = \text{MaxBlock}(H[j])$, which implies $S(j)$ (a). Since $\text{Seq}(D_{\text{special}}) = \text{Seq}(D_{\text{other}})$ in state $H[j]$, $S(j)$ (c) also follows.

Part (d): First, we will show that $y$ is the second argument of some $\text{chngCurBlock}$ operation execution in $H$. This is either $p$’s operation execution $Op$ under consideration, if step $j$ occurs at line 233 or line 250, or by Lemma 7.22 and the algorithm it is a $\text{chngCurBlock}(x, ...)$ operation execution $Op'$ by $p_{\text{special}}$ (that $Op$ is “helping”) if step $j$ occurs at line 256. Furthermore, in the latter case, by Lemma 7.23 $p_{\text{special}}$ does not change the state of $D_{\text{special}}$ or $D_{\text{other}}$ during $Op'$ (because $p$ does during $Op$). Thus, there is a distinct $\text{chngCurBlock}(..., y)$ operation execution in $H$ for every step in $H$ that changes the state of $D_{\text{special}}$ or $D_{\text{other}}$ to $(y, s)$ for some sequence number $s$. This and Condition 7.18 (c) imply part (d) since by Condition 7.18 (c), $y$ is unique and different from the initial block in each such operation execution.

We will now show that the implementation $I_{BM-DSM}$ is linearizable using the same approach as in Chapter 5. For each history $H$ of $I_{BM-DSM}$, we define a candidate lin-
earization $\bar{H}$ as follows. First, for each operation execution on the target object in $H$, we assign a “timestamp” of the form $(t, q)$ where $t$ is an integer and $q$ is a process ID or 0.

**Definition 7.25.** The timestamp $s$ for an arbitrary operation execution $Op$ in $H$, say by process $p$, and its completion (where applicable), are defined as follows:

**Operation types getCurBlock():**

(a) If $Op$ is complete in $H$ and returns at line 222, and $p$ reads $D_{special}$ at line 219 in step $i$ of $H$, then $s = (i, 0)$.

(b) If $Op$ is complete in $H$ and returns at line 224, and $p$ reads $D_{special}$ at line 219 in step $i'$ of $H$, then $s = (i, p)$ where $i$ is the smallest integer $\geq i'$ such that $D_{other}$ takes on the value read by $p$ at line 220 in state $H[i]$.

(c) Otherwise $s$ is undefined, and $Op$ does not appear in $\bar{H}$.

**Operation type chngCurBlock($x, y$):**

(d) If $p = p_{special}$, and the first write to $D_{special}$ (by $p$ during $Op$ at line 233 or by another process at line 256) during a $chngCurBlock(x, ...)$ operation execution has occurred in $H$, say in step $i$, then $s = (i, p)$.

(The completion of $Op$, if $Op$ is pending in $H$, returns $p$’s ID.)

(e) Else if $p \neq p_{special}$ and $p$ writes $D_{other}$ at line 250 in step $i$ of $H$, then $s = (i, 0)$.

(The completion of $Op$, if $Op$ is pending in $H$, returns $p$’s ID.)

(f) Else if $Op$ is complete, and $p$ reads $x \triangleright winner$ at line 242 or line 263 in step $i$ of $H$, then $s = (i, 0)$.

(g) Otherwise $s$ is undefined, and $Op$ does not appear in $\bar{H}$.

To construct $\bar{H}$, we arrange operation executions for which timestamps are defined, in increasing order of timestamp. We prove the uniqueness of timestamps in Lemma 7.26, and define their order in Definition 7.27.

**Lemma 7.26.** The timestamp of each operation execution in $H$ (for which the timestamp is defined) is unique.
Proof. This follows because, by Definition 7.25, if the timestamp of an operation execution $Op$ by $p$ in $H$ is $(t, q)$, then either $p$ executes step $t$ in $H$ during $Op$ and $q = 0$ (clauses other than (b) and (d)), or $Op$ is pending in state $H[t]$ and $q = p$ (clauses (b) and (d)). It remains to show why $Op$ is pending in state $H[t]$ in clauses (b) and (d). For clause (b), this follows directly from Definition 7.25. For clause (d), this follows from Condition 7.18 (b) and Lemma 7.22.

Definition 7.27. For timestamps $(t_1, p_1)$ and $(t_2, p_2)$, we say that $(t_1, p_1) < (t_2, p_2)$ if and only if $t_1 < t_2$, or $t_1 = t_2$ and $p_1 < p_2$.

Lemma 7.28. For any history $H$ of implementation $I_{BM-DSM}$, and any step $i$ in $H$, if a getCurBlock operation execution $Op$ by process $p$ in $H$ has timestamp $(i, ...)$ (see Definition 7.25 (a)–(b)), and $Op$ returns $x$, then $x = \text{MaxBlock}(H[i])$.

Proof. If the timestamp of $Op$ falls under Definition 7.25 (a), then $x = \text{Block}(D_{\text{special}})$ in state $H[i]$, and $Op$ returns $x$ at line 222. By the success of the test at line 221, $\text{Seq}(D_{\text{special}})$ in state $H[i]$ is greater than $\text{Seq}(D_{\text{other}})$ in some later state (when $p$ reads $D_{\text{other}}$), hence in state $H[i]$ by Lemma 7.24 (b). Thus, $x = \text{MaxBlock}(H[i])$, as wanted.

If the timestamp of $Op$ falls under Definition 7.25 (b), then $x = \text{Block}(D_{\text{other}})$ in state $H[i]$. Suppose $p$ reads $D_{\text{special}}$ at line 219 in step $j$ during $Op$, and $D_{\text{other}}$ at line 220 in step $j' > j$ during $Op$. If $D_{\text{other}}$ does not change state between $H[j']$ and $H[j]$, then $i = j$, and $x = \text{MaxBlock}(H[i])$ holds by the algorithm for getCurBlock. Otherwise, $(x, ...)$ is written to $D_{\text{other}}$ in step $i$ of $H$, and this changes the state of $D_{\text{other}}$, and so $x = \text{MaxBlock}(H[i])$ by Lemma 7.24 (a).

Lemma 7.29. $\bar{H}$ satisfies properties (a) and (b) of linearizability (sequential completion and order preservation).

Proof. Property (a) follows from our construction of $\bar{H}$ and Definition 7.25. For property (b), note that by Definition 7.25, if the timestamp of an operation execution $Op$ by $p$ in $H$ is $(i, ...)$, then $Op$ is pending in state $H[i]$ (see proof of Lemma 7.26). Thus, if $Op$ and $Op'$ are operation executions in $H$ such that $Op$ precedes $Op'$, then $Op$ has a smaller timestamp than $Op'$ by Definition 7.27, as wanted.

It remains to prove property (c) of linearizability (conformity to type $\tau_{BM}$). To that end, we first define $Op_i$, $s_i$, $p_i$, $x_i$, $y_i$, and $\nu_i = (C_i, L_i)$ as in Chapter 5.1.
Lemma 7.30. Implementation $I_{BM-DSM}$ satisfies property (c) of linearizability (conformity to type $\tau_{BM}$).

Proof. Let $H$ be any history of $I_{BM-DSM}$. Since conformity to a type is a safety property it suffices to consider finite $\bar{H}$. Let $k = |\bar{H}|$. Define $s_0 = (b_0, 0)$ and $s_{k+1} = (\infty, 0)$, where $b_0$ is the initial block. Let $(t_i, q_i)$ denote the timestamp $s_i$.

We will prove that for any $i \in \mathbb{N}$, $0 \leq i \leq k$:

(a) For $t = t_i$ and any integer $t \in [t_i, t_{i+1})$, $\text{MaxBlock}(H[t]) = C_i$.

(b) If $i > 0$, then the response of $Op_i$ is the correct response for an operation execution of that type applied in state $\nu_{i-1}$.

Part (b) implies the lemma, but we require both parts for induction. Now let $S(i)$ denote parts (a)–(b) for a particular value of $i$. Note that in $H$, the state of $D_{special}^\prime$ or $D_{other}^\prime$ (hence $\text{MaxBlock}$) is changed only by an execution of line 233, line 250, or line 256, which is an atomic step that defines the timestamp of an operation execution (on the target object) in $H$. For writes to $D_{special}$ this follows from Definition 7.25 (d), Lemma 7.23, and Lemma 7.22 (which implies that if a non-special process writes $D_{special}$ in step $t$ then there is an operation execution by $p_{special}$ in $\bar{H}$ with timestamp $(t, p_{special})$). Therefore, the state of $D_{special}$ and $D_{other}$ does not change between atomic steps $t_i$ and $t_{i+1}$ in $H$. This, in turn, implies that to prove part (a) of $S(i)$, it suffices to prove that $\text{MaxBlock}(H[t_i]) = C_i$ — and that is all we do in the inductive step that follows.

For $S(0)$, (a) follows from our earlier definition of $s_0 = (b_0, 0)$, and the initialization of $D_{special}$ and $D_{other}$ to $(b_0, 0)$. Part (b) holds trivially. Now for any $i$, $0 < i \leq k$, suppose that $S(i-1)$ holds, and consider $S(i)$. We proceed by cases on how the timestamp $s_i$ of $Op_i$ was obtained.

Case A: $Op_i$ falls under Definition 7.25 (a) or (b), and has timestamp $s_i = (t_i, q_i)$. In this case, $Op_i$ is a $\text{getCurBlock}$ operation execution, and so $C_i = C_{i-1}$.

It follows from Lemma 7.28 that $x = \text{MaxBlock}(H[t_i])$, and so to prove $S(i)$ (a) and $S(i)$ (b), it suffices to show that $C_{i-1} = x$. If step $t_i$ in $H$ is a read step by $p_i$, then $x = \text{MaxBlock}(H[t_i])$ implies $x = \text{MaxBlock}(H[t_i - 1])$, and $t_{i-1} < t_i$ holds by our construction of $\bar{H}$ (Definitions 7.25 and 7.27). Thus, $S(i-1)$ (a) implies $C_{i-1} = x$, as wanted. Otherwise, step $t_i$ in $H$ is a write step by some $q \neq p_i$, and so by our construction of $\bar{H}$ (Definitions 7.25 and 7.27) the operation execution
Op in $\bar{H}$ that immediately precedes $Op_i$ also has a timestamp of the form $(t_i, ...)$. Consequently, $\text{MaxBlock}(H[t_i]) = C_{i-1}$ holds by $S(i-1)$ part (a), and so $C_{i-1} = x$ since $\text{MaxBlock}(H[t_i]) = x$.

**Case B:** $Op_i$ falls under Definition 7.25 (f). In this case, $Op_i$ is a $\text{chngCurBlock}(x, y)$ operation execution for some $x$ and $y$, and $s_i = (t_i, 0)$ where step $t_i$ is a read of $x \triangleright winner$ by $p_i$. (We discharge this case before the remaining ones so that we can claim later on that any operation execution that falls under Definition 7.25 (f) is a failed $\text{chngCurBlock}$.)

If $p_i$ reads $x \triangleright winner$ at line 242 during the counterpart of $Op_i$ in $H$, then by the algorithm and Definition 7.25 (f), $p_i = p_{special}$ and $p_i$ completes line 240 before line 242, but does not write $x \triangleright winner$. Since $x \triangleright winner$ is initially $\bot$, $p_i$ completing line 240 implies that some non-special process $p'$ wrote $x \triangleright winner$ at line 251 or line 257. In fact, $p'$ must have executed line 251 otherwise by the algorithm $p$ lost $x \triangleright \text{LeaderElect}()$ at line 244, and so another non-special process $p''$ won $x \triangleright \text{LeaderElect}()$ by Lemma 7.19 and Specification 3.5, which implies that $p'$ and $p''$ both acquired the pseudo-lock in block $x$, contradicting Lemma 7.19 and Specification 4.2 (b). (Note that $p_i$ does not win $x \triangleright \text{LeaderElect}()$ at line 232 by the algorithm, our assumption that $p_i$ completes line 240, and Condition 7.18 (b).)

By our construction of $\bar{H}$ (Definitions 7.25 and 7.27), the execution of line 251 by $p'$ happens during the counterpart of a $\text{chngCurBlock}(x, ...)$ operation execution $Op'$ that precedes $Op_i$ in $\bar{H}$. Thus, $Op_i$ is a failed $\text{chngCurBlock}(x, y)$ operation execution, and so $C_i = C_{i-1}$.

$S(i)$ part (a) follows from $S(i-1)$ part (a) since $C_i = C_{i-1}$ and step $t_i$ does not write $D_{special}$ or $D_{other}$.

For $S(i)$ part (b), we must show that $Op_i$ returns the ID of the process $p_j$ that applies the earliest $\text{chngCurBlock}(x, ...)$ in $\bar{H}$. By Lemma 7.21, since $p'$ writes a process ID to $x \triangleright winner$ before step $t_i$, and by the algorithm for $\text{chngCurBlock}$, $Op'$ and $Op_i$ both return $p'$. Since $Op'$ is a $\text{chngCurBlock}(x, ...)$ that precedes $Op_i$ in $\bar{H}$, and its response is correct by $S(i-1)$ (b), the response of $Op_i$ is also correct.

**Case C:** $Op_i$ falls under Definition 7.25 (d). In this case, $Op_i$ is a $\text{chngCurBlock}(x, y)$ operation execution for some $x$ and $y$, $s_i = (t_i, p_i)$, and $p_i = p_{special}$.

First, we will show that $Op_i$ is the earliest $\text{chngCurBlock}(x, ...)$ operation execution
in $\bar{H}$. (Call this observation (⋆).) Suppose otherwise. By Definition 7.25 (d) and our analysis in Case B, the first $\text{chngCurBlock}(x,...)$ in $\bar{H}$, say $Op'$, falls under Definition 7.25 (d) or (e). Since $p_{\text{special}}$ executes $Op_i$ and we assume $Op'$ is not $Op_i$, some non-special process $p'$ executes $Op'$ by Condition 7.18 (b). Since $Op_i$ falls under Definition 7.25 (d), a write to $D_{\text{special}}$ occurs in the counterpart of some $\text{chngCurBlock}(x,...)$ in $H$, and so by Lemma 7.20 $p'$ does not write $D_{\text{other}}$ during the counterpart of $Op'$ in $H$. Since $p' \neq p_{\text{special}}$, this implies $Op'$ falls under Definition 7.25 (f), which contradicts our earlier observation that $Op'$ falls under Definition 7.25 (d) or (e). Thus, $Op_i$ is a successful $\text{chngCurBlock}(x,y)$ operation execution, and so $C_i = y$.

For $S(i)$ part (a), we must show that $\text{MaxBlock}(H[t]) = y$. By observation (⋆) and Definition 7.25 (d), the write to $D_{\text{special}}$ in step $t_i$ is the earliest such write during a $\text{chngCurBlock}(x,...)$, and so by Lemma 7.23 it changes the state of $D_{\text{special}}$. Thus, it follows from Lemma 7.24 (a) that $\text{MaxBlock}(H[t]) = y'$ where $y'$ is the block address embedded in the value written in step $t_i$. It remains to show that $y' = y$. If $p_i$ applies step $t_i$ (at line 233), then this occurs during the counterpart of $Op_i$ in $H$ by Condition 7.18 (b), and $y' = y$ follows directly from the algorithm for $\text{chngCurBlock}$ for $p_{\text{special}}$. Otherwise, by Condition 7.18 (b), Definition 7.25 (d), and the algorithm, some non-special process $p'$ applies step $t_i$ (at line 256) during a $\text{chngCurBlock}(x,...)$. Furthermore, by Lemma 7.22 and lines 255–256, $p'$ writes to $D_{\text{special}}$ the value $p_i$ wrote earlier to $x \triangleright A$ at line 231 during the counterpart of $Op_i$ in $H$. The block address embedded in this value is $y$, hence $y' = y$, as wanted.

For $S(i)$ part (b), we must show that $Op_i$ returns $p_i$’s ID. If the counterpart of $Op_i$ in $H$ is pending, then this follows from Definition 7.25 (d). Otherwise, $p_i$ executes lines 234 and 242 during the counterpart of $Op_i$ in $H$, and so by Lemma 7.21 and the algorithm, $Op_i$ returns $p_i$’s ID.

**Case D:** $Op_i$ falls under Definition 7.25 (e). In this case, $Op_i$ is a $\text{chngCurBlock}(x,y)$ operation execution for some $x$ and $y$, and $s = (t_i,0)$.

First, we will show that $Op_i$ is the earliest $\text{chngCurBlock}(x,...)$ operation execution in $\bar{H}$. (Call this observation (⋆).) Suppose otherwise. As argued in Case B, the first $\text{chngCurBlock}(x,...)$ in $\bar{H}$, say $Op'$, falls under Definition 7.25 (d) or (e). In the first case, there is a write to $D_{\text{special}}$ during a $\text{chngCurBlock}(x,...)$ in $H$, which contradicts Lemma 7.20 since there is a write to $D_{\text{other}}$ during the counterpart of
Op_i in H by Definition 7.25 (e). In the second case, there are writes to D_other in the counterparts of Op' and Op_i in H, which again contradicts Lemma 7.20 since we assume Op_i is not Op'. Thus, Op_i is a successful chngCurBlock(x, y) operation execution, and so C_i = C_{i-1}.

For S(i) part (a), we must show that MaxBlock(H[t_i]) = y, where y is the second argument of Op_i. By Definition 7.25 (e) and the algorithm for chngCurBlock, the block address embedded in the value written in step t_i is y. Moreover, this value has never been written to D_other by the algorithm and Condition 7.18 (c), and so the write in step t_i changes the state of D_other. Consequently, MaxBlock(H[t_i]) = y follows from Lemma 7.24 (a).

For S(i) part (b), we must show that Op_i returns p_i’s ID. If the counterpart of Op_i in H is pending, then this follows from Definition 7.25 (e). Otherwise, p_i executes lines 251 and 263 during the counterpart of Op_i in H, and so by Lemma 7.21 and the algorithm, Op_i returns p_i’s ID.

\[\square\]

**Lemma 7.31.** For any history H of implementation I_{BM-DSM}, and any getCurBlock or chngCurBlock operation execution Op in H, say by process p, the number of RMRs p incurs in H while executing Op in the DSM model is zero if p = p_{special}, and O(1) otherwise.

**Proof.** For p = p_{special}, note that p incurs zero RMRs executing the leader election algorithm in each block since this algorithm is local to p (i.e., satisfies Specification 3.9) by Lemma 7.19. Furthermore, any base object accessed by p_{special} outside of the LE algorithm is local to p_{special}. For p \neq p_{special}, this follows from the structure of the access procedures for getCurBlock and chngCurBlock, from the RMR complexity of the leader election algorithm and pseudo-lock in each block, and from Lemma 7.19. \[\square\]

**Lemma 7.32.** The implementation I_{BM-DSM} satisfies Specification 5.4 (termination).

**Proof.** Let H be any fair history of implementation I_{BM-DSM}. Each call to getCurBlock terminates in H by the structure of the access procedure. Next, consider a call to chngCurBlock in H. If a leader election algorithm is executed during this call, then it terminates by Specification 3.6 and Lemma 7.19. If a pseudo-lock is accessed, then
the functions \texttt{Pseudo-Enter} and \texttt{Pseudo-Exit} terminate by Lemma 7.19 and Specification 4.3, and since any process that acquires a pseudo-lock in any block \( x \) (at line 243) eventually calls \( x \triangleright \text{Pseudo-Exit}() \) (at line 261). Finally, we must show that any execution of the busy-wait loops at line 237 and line 240 terminates. Suppose, for contradiction, that some process loops forever in one of these.

\textbf{Case A:} line 237. Only process \( p_{\text{special}} \) may execute line 237, and if it reaches this line, then by the success of the test at line 236, it read \( x \triangleright \text{helping} = \text{true} \) earlier, and so some non-special process \( p' \) wrote \( x \triangleright \text{helping} = \text{true} \) at line 253 by the algorithm and initialization of \( x \triangleright \text{helping} \) to \text{false}. Since \( H \) is fair, \( p' \) eventually writes \( x \triangleright \text{helperDone} = \text{true} \) at line 259, and by the algorithm no process overwrites this variable with \text{false}. This contradicts \( p_{\text{special}} \) repeatedly reading \( x \triangleright \text{helperDone} = \text{false} \) at line 237.

\textbf{Case B:} line 240. Only process \( p_{\text{special}} \) may execute line 240, and if it reaches this line, then by the failure of the test at line 236, if did not win \( x \triangleright \text{LeaderElect}() \) earlier at line 232. Consequently, by Lemma 7.19, Specification 3.5, the algorithm, and the fairness of \( H \), some non-special process \( p' \) wins \( x \triangleright \text{LeaderElect}() \) at line 244. Since \( H \) is fair, \( p' \) eventually writes its ID to \( x \triangleright \text{winner} \) at line 251, and by the algorithm no process overwrites this variable with \text{⊥}. This contradicts \( p_{\text{special}} \) repeatedly reading \( x \triangleright \text{winner} = \text{⊥} \) at line 240.

\begin{proof}
\textbf{Theorem 7.33.} The implementation \( I_{BM-DSM} \) satisfies Specifications 6.1 and 6.2. Furthermore, each operation execution on the target object incurs \( O(1) \) RMRs in the DSM model. Finally, \( I_{BM-DSM} \) satisfies the locality property in the DSM model with respect to the designated process \( p_{\text{special}} \).

\textbf{Proof.} Specification 6.1 (linearizability) follows directly from Lemma 7.29 and Lemma 7.30. Specification 6.2 (termination) follows directly from Lemma 7.32. \( O(1) \) RMR complexity and locality follow from Lemma 7.31.
\end{proof}

The sequence numbers embedded in the registers \( D_{\text{special}} \) and \( D_{\text{other}} \) can grow without bound in \( I_{BM-DSM} \). We discuss this problem below in Section 7.3.1. The high-level idea is that since sequence numbers are non-decreasing and \text{MaxSeq} grows in small increments (see Lemma 7.24 (b)–(c)), they can be represented mod \( N \) and still compared correctly, with some modifications to the access procedures for \texttt{getCurBlock} and \texttt{chngCurBlock}.

To complete our analysis, we justify why Condition 7.18 holds when the implementation \( I_{BM-DSM} \) is used in conjunction with the ECAS implementation from Chapter 6.1.
Lemma 7.34. In implementation $I_E$ of ECAS from Chapter 6.1, the block manager base object $M$ is accessed according to Condition 7.18.

Proof.
Part (a): This follows directly from the structure of the access procedure for operation type ECAS.
Part (b): This follows from the definition of type $\tau_{BM}$, from Lemma 6.5, and from the structure of the access procedures, whereby at most one call to $\text{chngCurBlock}(x, y)$ occurs, and follows a call to $\text{getCurBlock}$ that returns $x$.
Part (c): This follows from Specification 5.1 since for each call to $\text{chngCurBlock}(x, y)$, the argument $y$ is obtained by first calling $\text{AllocBlock}$.

Theorem 7.35. The implementation $I_{E-DSM}$ of ECAS described in this section satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the DSM model. Finally, $I_{E-DSM}$ satisfies the locality property in the DSM model with respect to the designated process $p_{special}$.

Proof. Recall that $I_{E-DSM}$ is obtained from $I_E$ by

(a) declaring certain shared variables to be local to the designated process $p_{special}$, and
(b) changing the implementation of the block manager to $I_{BM-DSM}$ described in this section.

Specification 6.1 under Condition 6.4 follows from the corresponding properties of $I_E$ and $I_{BM-DSM}$ (Theorems 6.16 and 7.33). Specification 6.2 and $O(1)$ RMR complexity follow similarly. Locality of $I_{E-DSM}$ follows from the fact that, for any process $p$, (a) the shared objects accessed by $p$ other than the block manager $M$ are registers declared local to $p$; and (b) we assume that $M$ is obtained using the locally-accessible implementation $I_{BM-DSM}$ (see Theorem 7.33).

7.3.1 Bounding the sequence numbers

The sequence numbers embedded in the registers $D_{special}$ and $D_{other}$ can grow without bound in $I_{BM-DSM}$. To bound them, we take advantage of two properties of the implementation. First, as we showed in Lemma 7.24 (b)–(c), the sequence numbers grow only in small increments. Second, as long as process $p_{special}$ is not overwriting $D_{special}$,
the sequence numbers eventually stabilize because of the formula used by non-special processes to compute the next sequence number at lines 245–249. To take advantage of the second point, we make a slight modification to the access procedures that ensures process \( p_{\text{special}} \) can overwrite \( D_{\text{special}} \) at most once while a non-special process is between line 219 and 220 of \texttt{getCurBlock}.

The modification introduces the following global shared variables: Boolean arrays \( GB\text{WaitFlag}[1..N] \) and \( GB\text{Active}[1..N] \), all elements initialized to \texttt{false} and local to process \( p_{\text{special}} \). These arrays are used by non-special processes to indicate to \( p_{\text{special}} \) whether they are executing inside \texttt{getCurBlock}. When a non-special process \( p \) begins \texttt{getCurBlock}, it assigns \( GB\text{Active}[p] = \texttt{true} \), and then just before returning it assigns \( GB\text{Active}[p] = \texttt{false} \) and \( GB\text{WaitFlag}[p] = \texttt{true} \). Process \( p_{\text{special}} \), upon completing a call to \texttt{chngCurBlock}, resets \( GB\text{WaitFlag}[1..N] \) to \texttt{false}. It then loops through the elements of \( GB\text{Active}[1..N] \), and for each \( q \) such that \( GB\text{Active}[q] = \texttt{true} \), it waits for \( GB\text{WaitFlag}[q] = \texttt{true} \).

It is straightforward to show that these modifications preserve the termination, RMR complexity, and locality properties (see Theorem 7.33) of the implementation. Furthermore, it follows easily that for any process \( p \), while \( p \) is continuously between between line 219 and 220 of \texttt{getCurBlock}, process \( p_{\text{special}} \) may overwrite \( D_{\text{special}} \) at most once. This and Lemma 7.24 (b)–(c) imply that when a process reads \( D_{\text{special}} \) and \( D_{\text{other}} \) during one call to \texttt{getCurBlock}, it observes sequence numbers that differ by at most three. Thus, it suffices to store the modulo seven remainder of each sequence number, with small modifications to the tests at lines 221, 226 and 225 to compare such values correctly. This bounds the number of bits needed to represent the sequence numbers at \( O(1) \).

### 7.4 Necessity and Sufficiency of Locality Properties

We now discuss the necessity and sufficiency of our locality properties, with respect to the RMR preservation property.

#### 7.4.1 DSM Model

Although we have not formalized an RMR preservation property for the DSM model, the natural formalization is the analog of Definition 7.1 where “CC model” is replaced by “DSM model”, and we fix \( c_2 = 0 \) in the case when \( p \) is the designated process \( p_{\text{special}} \).
to which \( I \) is local. (We fix \( c_2 = 0 \) because the designated process must not perform any RMRs at all in \( H \), even if it has a pending operation execution in \( H \) that is discarded from \( \bar{H} \).) It is easy to see that the locality property in the DSM model, combined with worst-case \( O(1) \) RMR complexity, implies this RMR preservation property. In that sense, we consider the locality property sufficient for RMR preservation. The locality property in the DSM model is also necessary for RMR preservation. To see why, consider an algorithm that uses a hardware-implemented CAS object \( C \) local to some designated process \( p_{\text{special}} \). Suppose that all other objects accessed by \( p_{\text{special}} \) are local to \( p_{\text{special}} \). Thus, the RMR complexity of the algorithm for process \( p_{\text{special}} \) is zero. If we replace \( C \) with a software implementation that does not satisfy the locality property in the DSM model, then some operation execution applied to \( C \) by \( p_{\text{special}} \) may incur one or more RMRs, or the RMR complexity of the algorithm for some non-special process may increase by more than a constant factor. In either case, the RMR preservation property for the DSM model breaks.

### 7.4.2 CC Model

In the CC model, the RMR preservation property was stated earlier in Definition 7.1. The locality properties in the write-through and write-back CC model, combined with worst-case \( O(1) \) RMR complexity, imply the RMR preservation property for an implementation. To see this, first consider the simple write-through model. Consider a history \( H \) of an algorithm \( \mathcal{A} \) where some process \( p \) incurs \( k \) RMRs accessing some hardware-implemented shared object \( C \). (If \( p \) incurs an unbounded number of RMRs then the RMR preservation property holds trivially.) We can think of \( p \)'s accesses to \( C \) in \( H \) as a series of \( k \) execution subhistories \( H_1, H_2, \ldots, H_k \), such that for \( 1 \leq i \leq k \), \( H_i \) contains one or more accesses to \( C \) by \( p \) and only the first access to \( C \) in \( H_i \) causes an RMR (i.e., any access after the first is done “in-cache”). It follows from the definition of \( H_i \) that no process applies a write-like operation on \( C \) between \( p \)'s first and last access to \( C \) in \( H_i \). Consequently, if we replace \( C \) with a software implementation that is locally-accessible in the write-through CC model, then locality property (R) tells us that \( p \) incurs \( O(1) \) RMRs executing \( H_i \), for any \( i \). Thus, \( p \) incurs \( O(k) \) RMRs in total accessing (the software implementation of) \( C \), which amounts to a constant-factor increase in RMR complexity, as needed for the RMR preservation property.

The analysis for the write-back model is similar. We define \( H_1, H_2, \ldots, H_k \) in the same
way, and suppose that $C$ is replaced with a locally-accessible software implementation. For each $H_i$, we can deduce that $p$ now incurs $O(1)$ RMRs executing the operations on $C$ in $H_i$, using either locality property (R) (see Definition 7.2) or locality property (W) (see Definition 7.3). To decide which property applies, we look at the status of the cached copy of $C$ on which $p$ operates in $H$. The status immediately following each step in $H_i$ is either “shared” or “exclusive” and does not change throughout $H_i$, otherwise that would imply an RMR by $p$ as a result of an operation on $C$ by $p$ that occurs between the first and last step of $C$, which contradicts the definition of $H_i$. If the status is “shared”, then it follows that all of $p$’s operations in $H_i$ are read-like, and appropriate conditions hold to apply property (R), as in the analysis of the write-through model. Otherwise, the status is “exclusive”, which means that no process other than $p$ accesses $C$ between the first and last step of $C$. In that case, property (W) applies. In either case, if we replace $C$ with a software implementation that is locally-accessible in the write-back CC model, then $p$ incurs $O(k)$ RMRs in total accessing $C$, as wanted.

So far we have shown that the locality properties in the CC model are sufficient for RMR preservation. The same properties are not necessary for RMR preservation. Intuitively, this is because the locality properties bound the number of RMRs corresponding to some subsequence of operations on the target object, whereas the RMR preservation property refers to the total number of RMRs performed by a process in a history. Consequently, even if the number of RMRs incurred by certain operations is “over-budget” (e.g., $N$ instead of $O(1)$), it is still possible that the number of RMRs a process incurs in an entire history increases by only a constant factor, for example because the process has already incurred many RMRs before such operations. Thus, it is possible to weaken the locality properties, by allowing some subsequences of operations to incur more than $O(1)$ RMRs, while making the properties strong enough to ensure at most a constant-factor increase in RMR complexity for the history as a whole.

### 7.5 Conclusion

In this chapter we defined certain “locality properties” needed for our implementations of synchronization primitives to simulate the behaviour of their hardware counterparts in terms of RMR complexity (and not only linearizability). Thus, we have proved our principal result (2) from Chapter 1. Before we can prove result (3), we must also show how to make our implementations writable, while preserving the locality properties. We
do this in Chapter 8.

The locally-accessible implementations presented in this chapter are much more complex conceptually than their counterparts from Chapter 6. For some operation executions this increase in conceptual complexity corresponds also to an increase in the number of RMRs incurred. Consequently, it is natural to ask whether it is possible to obtain the RMR preservation property (see Definition 7.1) from properties weaker than those used in this chapter. The latter properties are: $O(1)$ RMR complexity per operation execution and a locality property specific to a particular shared memory model.

As regards $O(1)$ RMR complexity, we considered this property in order to prove our principal result (1), which is tied closely to our ranking of synchronization primitives. For result (3) (which refers to the RMR complexity of entire histories rather than individual operation executions), we do not need $O(1)$ RMR complexity and so the RMR complexity property can be relaxed. For example, if a process invokes $k$ operation executions on the implemented object and the first $k-1$ together incur $\Theta(k)$ RMRs, then the $k$’th one alone may incur as many as $\Theta(k)$ RMRs without inflating the RMR complexity of the entire history by more than a constant factor. We discuss how to exploit this observation in Chapter 13.

As regards the locality property, we explained in Section 7.4 that this property is necessary in the DSM model but not in the CC model. We leave open the question of how the locality property can be weakened in the CC model so that our locally-accessible implementations can be simplified.
Chapter 8

Writable Implementations of ECAS

In this chapter, we describe a writable implementation of ECAS that builds on the techniques introduced in Chapters 6 and 7. A writable ECAS is a type like ECAS except that, in addition, it supports operation type \texttt{Write} whose transition relation is defined by (the atomic execution of) the pseudo-code shown in Figure 8.1. Let \( \tau_{\text{ECAS-W}} \) denote this type.

<table>
<thead>
<tr>
<th>Function \texttt{Write}(\textit{new})</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Input:} \textit{new} – value to be stored</td>
</tr>
<tr>
<td>( V := \textit{new} )</td>
</tr>
<tr>
<td>( \textbf{foreach} \ i \in 1..N \ \textbf{do} \ \text{Linked}[i] := \texttt{false} )</td>
</tr>
<tr>
<td>( \textbf{return} \ \texttt{OK} )</td>
</tr>
</tbody>
</table>

Figure 8.1: Definition of \texttt{Write} operation type for type \( \tau_{\text{ECAS-W}} \). (The current state is denoted by \( V \) and \( \text{Linked}[1..N] \).)

We now present an implementation \( I_{EW} = (\tau_{\text{ECAS-W}}, \mathcal{P}, \mathcal{B}, \mathcal{H}) \) that is similar in spirit to the non-writable one presented in Chapter 6. We rely on a block manager, and define blocks so that each contains a base object \( B \) of non-writable ECAS type, as well as a register \texttt{writer} that records the ID of the process that made a particular block current. The access procedures for the operation types \texttt{Read}, \texttt{LL}, \texttt{ECAS} and \texttt{Write} of \( I_{EW} \) are shown in Figure 8.2. Lines containing \textbf{shaded} statements can be ignored safely for now; these statements come into play in Chapter 8.2 when we discuss locally-accessible implementations. For now, the subroutines \texttt{HelperBegin}, \texttt{HelperEnd} and \texttt{HelperCC} have trivial implementations (see Figure 6.7).
The access procedures for operation types \texttt{Read}, \texttt{LL} and \texttt{ECAS} are obtained by applying the corresponding operation to the underlying non-writable ECAS object \((B)\) in the current block. The \texttt{Write(new)} operation execution attempts to change the current block to one where \(B\) is initialized to \texttt{new}, by applying a \texttt{chngCurBlock(d, d')} on the block manager at line 274. At this point, processes applying \texttt{Write} concurrently compete to decide whose \texttt{Write} operation execution will “succeed,” here meaning that it will be the last among the competing group in the linearization order. The effect of the “successful” operation execution becomes visible to subsequent operation executions, whereas the effects of the others are “overwritten” by the successful one.

One subtle point regarding the implementation that requires further explanation is the initialization of the base object \(d \triangleright B\) at line 273. A base object that is provided in hardware can be initialized by writing to it, but in this thesis we assume that \(d \triangleright B\) is an object implemented in software. The only implementations given so far for this object are the ones from Chapters 6–7, which are not writable. However, it is straightforward to provide an initialization operation type in those implementations because of the following simplifying observations: (1) The initialization of \(d \triangleright B\) is the first operation applied to this object. (2) The initialization is permitted to incur \(O(1)\) RMRs because the \texttt{Write} operation execution that calls it at line 273 already incurs several RMRs. (3) At most one process at a time accesses \(d \triangleright B\) at line 273, namely the process \(p\) that allocated the block \(d\) (see Specification 5.1).

The procedure for initializing \(d \triangleright B\) to a value \(val\) (which is implicit in our pseudo-code) works as follows: the calling process \(p\) simply writes \(val\) into the field \(V\) in the initial block in the implementation of \(d \triangleright B\) (see Chapters 6–7). (Recall that the implementation of \(d \triangleright B\) internally uses blocks that are distinct from those in Figure 8.2.) Successive applications of this initialization procedure can be used to re-initialize \(B\) – something we require later on in Section 8.2.1 (see line 296 in Figure 8.3 and Lemma 8.12). The main caveat is that when we re-initialize \(d \triangleright B\) to \(val\), we must ensure not only that \(V = val\) but also that \(\text{Linked}[1..N] = \text{false}\) in the current block underlying this object. Fortunately, this can be done using \(O(1)\) RMRs because each time we re-initialize \(d \triangleright B\), the current block in the implementation of \(d \triangleright B\) is still the initial block of that implementation, and furthermore all elements of \(\text{Linked}[1..N]\) other than possibly \(\text{Linked}[p]\) are still \text{false}. (This is because we re-initialize \(d \triangleright B\) only if it has been accessed by no process other than \(p\).)
Declarations

Shared variables: (global)

\( M \) – block manager from Chapter 5, initialized to the address of a fresh block

Shared variables: (per-block)

\( B \) – instance of \( O(1)\)-RMR non-writable ECAS, initialized to the initial state of type \( I_{EW} \) (where \( V \) can be any value and \( \text{Linked}[1..N] = \text{false} \))

\( \text{writer} \) – register, stores process ID or \( \perp \), initially \( \perp \)

Subroutines: (per-block)

\( \text{AllocBlock()} \) – block allocator from Chapter 5

Private variables: (per-process)

\( d, d' \) – block addresses, uninitialized

\( \text{ret} \) – Boolean, initially \( \text{false} \)

Function \( \text{Write}(val) \)

267 \( d := M.\text{getCurBlock}() \)
268 \( \text{HelperBegin}(d) \)
269 \( \text{ret} := \text{HelperCC}(d, val) \)
270 \( \text{if } \text{ret} = \text{false} \text{ then} \)
271 \( \text{/* Try to change current */} \)
272 \( d' := \text{AllocBlock()} \)
273 \( \text{write } d' \triangleright \text{writer} := \text{PID} \)
274 \( \text{initialize } d' \triangleright B \text{ to a state where} \)
275 \( \text{V} = \text{val} \text{ and } \text{Linked}[1..N] = \text{false} \)
276 \( M.\text{chngCurBlock}(d, d') \)
277 \( \text{HelperEnd}(d) \)
278 \( \text{return } \text{OK} \)

Function \( \text{ECAS}(isSC, cmp, new) \)

278 \( d := M.\text{getCurBlock}() \)
279 \( \text{HelperBegin}(d) \)
280 \( \text{ret} := (d \triangleright B).\text{ECAS}(isSC, cmp, new) \)
281 \( \text{HelperEnd}(d) \)
282 \( \text{return } \text{ret} \)

Function \( \text{Read()} \)

283 \( d := M.\text{getCurBlock}() \)
284 \( \text{HelperBegin}(d) \)
285 \( \text{ret} := (d \triangleright B).\text{Read()} \)
286 \( \text{HelperEnd}(d) \)
287 \( \text{return } \text{ret} \)

Function \( \text{LL()} \)

288 \( d := M.\text{getCurBlock}() \)
289 \( \text{HelperBegin}(d) \)
290 \( \text{ret} := (d \triangleright B).\text{LL()} \)
291 \( \text{HelperEnd}(d) \)
292 \( \text{return } \text{ret} \)

Figure 8.2: Implementation \( I_{EW} \) of writable ECAS.
8.1 Analysis

We proceed as in Section 6.1.1.

**Lemma 8.1.** The analog of Lemma 6.5 for $I_{EW}$ holds.

*Proof.* This follows by a proof analogous to the one given in Chapter 6.1. References to line 171 and line 175 are replaced by references to line 271 and line 274. □

**Lemma 8.2.** The analog of Lemma 6.6 for $I_{EW}$ holds.

*Proof.* This follows by the same proof as given in Chapter 6.1, with any reference to Lemma 6.5 replaced by a reference to Lemma 8.1. □

Now consider linearizability. For any history $H$ of the implementation, we define a candidate linearization $\bar{H}$ using the same general approach for (non-writable) ECAS in Chapter 6.1.1.

**Definition 8.3.** The timestamp $s$ for an arbitrary operation execution $Op$ in $H$, say by process $p$, and its completion (where applicable), are defined as follows:

**Operation type ECAS:** (and similarly for Read and LL, which are implemented in an analogous way)

(a) If $p$ executes $M.getCurBlock()$ at line 278 during $Op$, say with response $x$, and accesses $x \triangleright B$ at line 280 in step $i$ of $H$, then $s = (x, i, 0)$.

(If $Op$ is pending in $H$, its completion returns the value returned by the base object atomic step on $x \triangleright B$ in step $i$.)

(b) Otherwise, $s$ is undefined.

**Operation type Write:**

(c) If $p$ executes a successful $M.chngCurBlock(x, y)$ at line 274 during $Op$ in step $i$ of $H$, then $s = (y, i, 0)$. (If $Op$ is pending in $H$, its completion returns OK.)

(d) Else if $Op$ is complete in $H$, and $p$ executes a failed $M.chngCurBlock(b, y)$ at line 274 during $Op$, and a successful $M.chngCurBlock(b, x)$ (by any process) occurs in step $i$ of $H$, then $s = (x, i, -p)$. (Block $x$ is well-defined by the specification of the block manager type.)
(e) Otherwise, $s$ is undefined.

Next, we define $s_i = (x_i, t_i, ...), Op_i, p_i,$ and the candidate linearization $\bar{H}$ as in Section 6.1.1. (In particular, we continue to order timestamps according to Definition 6.9.)

**Lemma 8.4.** The analogs of Lemma 6.10 and Lemma 6.13 (properties (a) and (b) of linearizability – sequential completion and order preservation) for $I_{EW}$ hold.

**Proof.** Both lemmas follow for $I_{EW}$ by proofs analogous to those given in Chapter 6.1. References to Lemmas 6.5 and 6.6 are replaced by references to Lemmas 8.1 and 8.2. References to Definition 6.8 are replaced by references to Definition 8.3. The analogs of Definition 6.8 (d)–(e) are Definition 8.3 (c)–(d). The analogs of lines 160 and 175 are lines 267 and 274. □

**Lemma 8.5.** The analog of Lemma 6.15 (property (c) of linearizability – conformity to type $\tau_{ECAS-W}$) for $I_{EW}$ holds.

**Proof.** Since conformity to a type is a safety property it suffices to consider finite $\bar{H}$. Let $k = |\bar{H}|$. Define $x_0, x_{k+1}, t_0$ and $t_{k+1}$ as in the proof of Lemma 6.15. We will show that for any $i \in \mathbb{N}, 0 \leq i \leq k$:

(a) If the timestamp $t_i$ does not fall under Definition 8.3 (d), then for any integer $t \in [t_i, t_{i+1}), x_i \triangleright B = \nu_i$ holds in state $H[t]$.

(b) If $i > 0$, then the response of $Op_i$ is the correct response for an operation execution of that type applied in state $\nu_{i-1}$.

Part (b) implies the lemma, but we require both parts for induction. Now let $S(i)$ denote parts (a)–(b). Note that in $H$, the current block and state of $B$ in that block are changed only by an execution of line 274, line 280, line 285, or line 290, which is an atomic step that defines the timestamp of an operation execution on the target object in $\bar{H}$. Therefore, the current block and state of $B$ in that block do not change between atomic steps $t_i$ and $t_{i+1}$ in $H$. This, in turn, implies that to prove part (a) of $S(i)$, it suffices to prove that $x_i \triangleright B = \nu_i$ in state $H[s_i]$ – and that is all we do in the inductive step that follows.

For $S(0)$, part (a) follows from our earlier definition of $x_0$ as the initial block and $t_0 = 0$, as well as the initialization of $x_0 \triangleright B$ to the initial state of type $I_{EW}$. Now suppose that $S(i-1)$ holds for some $i, 0 < i \leq k$, and consider $S(i)$. We proceed by
cases on how $s_i = (x_i, t_i, \ldots)$ was obtained, noting that $x_i = x_{i-1}$ except possibly when $s_i$ falls under Definition 8.3 (c) or (d).

Case A: $Op_i$ falls under Definition 8.3 (a). In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, $Op_i$ is a Read or LL or ECAS operation execution, and $p_i$ applies the corresponding operation to $x_i \triangleright B$ in step $t_i$.

For $S(i)$ (a), first note that $x_i = x_{i-1}$ and that $Op_{i-1}$ does not fall under Definition 8.3 (d), otherwise by our construction of $\bar{H}$, $Op_i$ would be a Write operation execution falling under Definition 8.3 (c) or (d). Thus, it follows from $S(i-1)$ (a) that $x_i \triangleright B = \nu_{i-1}$ in state $H[t_i - 1]$. Consequently, $p_i$’s operation in step $t_i$ changes the state of $x_i \triangleright B$ to $\nu_i$, as wanted.

For $S(i)$ (b), note that $Op_i$ returns the response of step $t_i$, where $p_i$ applies an operation execution of the same type as $Op_i$ to $x_i \triangleright B$. Since we showed that $x_i \triangleright B = \nu_{i-1}$ in state $H[t_i - 1]$, this response is correct for $Op_i$ in $H$.

Case B: $Op_i$ falls under Definition 8.3 (c). In this case, $s_i = (x_i, t_i, 0)$ for some $t_i$, $Op_i$ is Write, and $p_i$ executes a successful $M.chngCurBlock(\ldots, x_i)$ at line 274 in step $t_i$ of $H$.

$S(i)$ (a) follows by the action of step $t_i$ by $p_i$ in $H$, which makes $x_i$ the current block, and from the prior initialization of $x_i \triangleright B$ to $val_i$ (i.e., the argument of $Op_i$) at line 273.

$S(i)$ (b) holds since $Op_i$ returns OK at line 277.

Case C: $Op_i$ falls under Definition 8.3 (d).

Here $S(i)$ (a) holds trivially since $Op_i$ falls under Definition 8.3 (d).

$S(i)$ (b) holds since $Op_i$ returns OK at line 277.

\[ \square \]

**Theorem 8.6.** The implementation $I_{EW}$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the CC and DSM models.

**Proof.** Specification 6.1 under Condition 6.4 follows from Lemma 8.4 and Lemma 8.5. RMR complexity and Specification 6.2 follow from the same arguments as in the proof of Theorem 6.16. \[ \square \]
8.2 Locality

In this section we describe, for each of the shared memory models under consideration in this thesis, how to transform the writable implementation $I_{EW}$ to an implementation $I'_{EW} = (τ_{ECAS-W}, P, B, H)$ that satisfies the locality property in that model.

8.2.1 CC Model

In the CC model with write-through and write-back caching, we construct $I'_{EW}$ from $I_{EW}$ by making the following modifications. First, we assume that the block manager and non-writable ECAS base object $B$ (in each block) are implemented as described in Chapter 5 and Section 7.2, respectively. Second, we override the subroutines HelperBegin, HelperEnd, and HelperCC in the same way as in Section 7.2, except for a slight change in HelperCC: In function HelperCC, we replace the Write operation on the register $V$ (see line 202 in Figure 7.1) with a statement that re-initializes the base object $B$, which is the analog of $V$ here. As we show later (see Lemma 8.10), only the process that allocated block $x$ will perform this for block $x$, and only before any other process has accessed $x \triangleright B$, which means that we can apply the special initialization operation discussed earlier. The modified implementation of HelperCC is shown in Figure 8.3.

```
Declarations
Private variables: (per-process)
  ret – Boolean, uninitialized

Function HelperCC(d, val)
  ret := false
  if read(d \triangleright writer) = PID then
    if TryToReuseBlock(d) = true then
      re-initialize d \triangleright B to a state where V = val and Linked[1..N] = false
      ret := true
    end
  end
  DoneReusingBlock(d)
  return ret
```

Figure 8.3: Subroutine HelperCC for locally-accessible writable ECAS implementation in the CC model.
In HelperCC, a process attempts to perform the Write operation execution by changing the state of $B$ in block $d$. If the process $p$ executing the Write is the process that allocated $d$ and no other process has “seen” block $d$ (i.e., the tests at lines 294 and 295 both succeed), then $p$ does not change the current block. Rather, it re-initializes $d \triangleright B$ to the argument new of its Write (line 296); in this case, HelperCC returns true. Otherwise (i.e., if $p$ is not the process that allocated $d$, or some process has “seen” $d$), $p$ proceeds as before: it allocates a new block $d'$, initializes $d' \triangleright B$ to new, and attempts to change the current block to $d'$ (lines 271–274 of Write).

**Lemma 8.7.** The analog of Lemmas 6.5 and 7.5 for $I'_{EW}$ holds.

*Proof.* This follows by the same proof as given in Chapter 6.1. □

**Lemma 8.8.** The analog of Lemmas 6.6 and 7.6 for $I'_{EW}$ holds.

*Proof.* This follows by the same proof as given in Chapter 6.1, with any reference to Lemma 6.5 replaced by a reference to Lemma 8.7. □

**Lemma 8.9.** The analog of Lemma 7.7 holds for $I'_{EW}$.

*Proof.* This follows by a proof analogous to the one given in Chapter 7.2 since $I'_{EW}$ uses the same implementations of subroutines TryToReuseBlock, DoneReusingBlock, HelperBegin and HelperEnd as $I_{EW}$, and since these functions are called in an analogous manner. Any reference to Lemma 7.5 in the proof is replaced by a reference to Lemma 8.7. □

**Lemma 8.10.** For any history $H$ of implementation $I'_{EW}$, any process $p$, and any block $x$, if $p$ is at line 296 during a call to HelperCC($x$, val) then:

(a) $p$ is the only process that has accessed $x \triangleright B$; and

(b) $x$ is current from the point when $p$ last called $M$.getCurBlock() until $p$ makes a call to DoneReusingBlock($x$) at line 299.

*Proof.* Note that if $p$ is at line 296 during a call to HelperCC($x$, val), then by the test at line 294, $p$ is the process that allocated $x$. Furthermore, $p$ has completed a call to TryToReuseBlock($x$) that returned true, and $p$ has not subsequently made a call to DoneReusingBlock($x$). Consequently, no process $q \neq p$ has completed a call to HelperBegin($x$) by Lemma 8.9 (specifically the analog of Lemma 7.7 (a) for $I'_{EW}$).
Since a process must complete a call to $\text{HelperBegin}(x)$ before accessing $x \triangleright B$, this implies part (a).

It follows similarly that no process $q \neq p$ completes a call to $\text{HelperBegin}(x)$ until $p$ makes a call to $\text{DoneReusingBlock}(x)$ at line 299. This implies part (b) because no process applies a successful $\text{M.chngCurBlock}(x, ...)$ between $p$’s last call to $\text{M.getCurBlock()}$, which returns $x$, and $p$’s subsequent call to $\text{DoneReusingBlock}(x)$ (if it occurs); $p$ itself does not do this by the algorithm, and no $q \neq p$ does so because by Lemma 8.8 that would imply $q$ calls $\text{M.chngCurBlock}(x, ...)$, which can only happen after $q$ completes a call to $\text{HelperBegin}(x)$.

**Lemma 8.11.** The analogs of Lemma 6.10 and Lemma 6.13 (properties (a) and (b) of linearizability – sequential completion and order preservation) for $I'_{EW}$ hold.

**Proof.** Both lemmas follow for $I'_{EW}$ by the same proofs as in Chapter 6.1, with reference to Lemmas 6.5 and 6.6 replaced by references to Lemmas 8.7 and 8.8.

To prove linearizability, we define for any history $H$ of $I'_{EW}$ a candidate linearization $\bar{H}$ as in Section 8.1, except that we augment the definition of timestamps (Definition 8.3). That is, we add a new clause (between clause (d) and clause (e)) for a $\text{Write}$ operation execution $Op$ in $H$ where line 296 is reached:

(g) Else if $p$ re-initializes $x \triangleright B$ at line 296 for some block $x$ during $Op$ in step $i$ of $H$,

then $s = (x, i, 0)$. (If $Op$ is pending in $H$, its completion returns OK.)

**Lemma 8.12.** The analog of Lemma 8.5 (property (c) of linearizability – conformity to type $\tau_{ECAS-W}$) for $I'_{EW}$ holds.

**Proof.** We modify the proof of Lemma 8.5 by extending the case analysis as follows:

**Case D:** $Op_i$ falls under Definition 8.3 (g). In this case, $Op$ is a $\text{Write}$ operation execution where $p_i$ re-initializes $x_i \triangleright B$ in step $t_i$ at line 296.

It follows from Lemma 8.10 (b) that $x_i$ is current in state $H[t_i - 1]$. Consequently, $S(i)$ (a) follows from the action of $p_i$’s step at time $t_i$. This step ensures that $(x_i \triangleright B).V = \text{val}$ in state $H[t_i]$ by re-initializing the value of $x_i \triangleright B$. Similarly, it ensures that $(x_i \triangleright B).\text{Linked}[p_i] = \text{false}$ in state $H[t_i]$. (Recall our prior explanation of the re-initialization operation at the end of Chapter 8.) As for the other elements of $(x_i \triangleright B).\text{Linked}[1..N]$, these are all $\text{false}$ in states $H[t_i - 1]$ and $H[t_i]$ by the
initialization of \( x_i \sqsupset B \), by Lemma 8.10 (a), and since the re-initialization operation execution does not write them.

\( S(i) \) (b) holds because \( O_{p_i} \) returns \( OK \).

**Lemma 8.13.** The analog of Lemma 7.12 (termination) for \( I'_{EW} \) holds.

*Proof.* This follows by a proof analogous to the one given in Chapter 7.2. References to Lemma 7.7 are replaced by references to Lemma 8.9.

**Lemma 8.14.** The analog of Lemma 7.13 (\( O(1) \) RMR cost for a process to access the block manager and allocator) for \( I'_{EW} \) holds.

*Proof.* This follows by a proof analogous to the one given in Chapter 7.2. References to Lemma 7.5 are replaced by references to Lemma 8.7.

**Lemma 8.15.** For any history \( H \) of \( I'_{EW} \), for any block \( x \) accessed in \( H \), and for any process \( p \), the number of RMRs that \( p \) incurs while accessing block \( x \) in \( H \), not including the field \( B \), is:

- \( O(1) \) in the CC model with write-back caching; and
- \( O(1 + m) \) in the CC model with write-through caching, where \( m \) is the number of write-like operation executions in \( H \) on the target object during which \( p \) accesses block \( x \).

*Proof.* This lemma is the counterpart of Lemma 7.14 for \( I'_{EW} \), but not its analog, because we ignore RMRs incurred while accessing \( B \), which is the counterpart of \( V \) in Section 7.2. (The analog does not hold because a process may incur arbitrarily many RMRs in a history where only block \( x \) is accessed.) Still, the lemma follows by an analogous proof, where we drop the case that considers the block field \( V \). References to Lemma 7.7 are replaced with references to Lemma 8.9.

**Lemma 8.16.** Implementation \( I'_{EW} \) satisfies the locality property in the write-through and write-back CC model (see Definitions 7.2 and 7.3).

*Proof.* Consider any history \( H \) of \( I'_{EW} \), and consider the linearization \( \bar{H} \) of \( H|O_\tau \) defined in our proof linearizability (see Lemma 8.12), where \( O_\tau \) is the target object. To prove the locality property, we will show that \( p \) incurs \( O(1) \) RMRs in \( H \) while executing the
counterparts of certain operation executions in $\bar{H}$, as in in the proof of Lemma 7.15. (As before, the case when $p$ does not access any block during some operation execution is discharged easily.) To that end, we will break down the analysis into two parts: accesses to the base object $B$ in blocks, and accesses to all other shared objects. 

**Property (R)** (Definition 7.2). We must consider the write-through and write-back CC models. Fix process $p$ and a sequence $\bar{H}'$ of consecutive read-like operation executions in $\bar{H}$. Let $H'$ denote the sequence of atomic steps (which access base objects) in $H$ corresponding to $\bar{H}'$.

First, we will show that $p$ accesses at most one block, say $x$, in $H'$. Suppose otherwise. By our definition of $\bar{H}'$ and $H'$, $p$ does not call $M.chngCurBlock$ or $AllocBlock()$ in $H'$, since that can only occur during a Write operation execution. Furthermore, calls to $M.getCurBlock()$ by $p$ in $H'$ return at most one block, otherwise between two such calls by $p$ there is a successful $M.chngCurBlock$ in $H$, and so there is a Write operation execution in $\bar{H}$ that is linearized between $p$’s first and last operation execution in $\bar{H}'$, contradicting the assumption that $\bar{H}'$ contains read-like operation executions only. Thus, by Lemma 8.14 and Lemma 8.15, $p$ incurs $O(1)$ RMRs accessing shared objects other than $B$ in $H'$.

It remains to consider $B$. Note that between any two base object atomic steps on $x \triangleright B$ by $p$ in $H'$, there is no write-like operation on $x \triangleright B$ by any other process, otherwise again we reach a contradiction since there is a write-like ECAS operation execution in $\bar{H}$ that is linearized between $p$’s first and last operation execution in $\bar{H}'$. Thus, by locality property (R) of $x \triangleright B$, $p$ incurs $O(1)$ RMRs accessing $x \triangleright B$ in $H'$, as wanted.

**Property (W)** (Definition 7.3). We need only consider the write-back CC model. Fix process $p$ and a sequence $\bar{H}'$ of consecutive operation executions by $p$ in $\bar{H}$. Again, let $H'$ denote the sequence of atomic steps in $H$ corresponding to $\bar{H}'$.

First, we will show that $p$ tries to change the current block at most once in $H'$. Suppose, for contradiction, that $p$ does this in the counterparts of operation executions $Op$ and $Op'$ in $\bar{H}'$. Arguing as in the proof of Lemma 7.15, property (W) (with references to Lemma 7.7 replaced by references to Lemma 8.9), this implies that there is an operation execution in $\bar{H}'$ between $Op$ and $Op'$ by a process different from $p$, which contradicts the definition of $\bar{H}'$. Next, note that calls to $M.getCurBlock$ by $p$ in $H'$ return at most two distinct values. This is because if three values are returned, then there are at least two successful calls to $M.chngCurBlock$ in $H$ between the first and last step in $H'$, where at most one is by $p$ (as argued above), and the other (by a process different from $p$) coincides
with the timestamp of a Write that appears between the first and last operation execution in $\bar{H}'$, which contradicts $\bar{H}'$ containing operation executions by $p$ only. Thus, $p$ accesses at most three blocks in $H'$, and so by Lemma 8.14 and Lemma 8.15, $p$ incurs $O(1)$ RMRs accessing shared objects other than $B$ in $H'$.

It remains to consider $B$. Note that for any block $x$ process $p$ accesses, and between any two base object atomic steps on $x \triangleright B$ by $p$ in $H'$, there is no atomic step at all on $x \triangleright B$ by any other process. To see this, note that otherwise by our construction of $\bar{H}$ there would be an ECAS, LL, or Read operation execution in $\bar{H}$ by a process different from $p$ that is linearized between $p$’s first and last operation execution in $\bar{H}'$, which contradicts $\bar{H}'$ containing operation executions by $p$ only. Thus, by locality property (W) of $x \triangleright B$, $p$ incurs $O(1)$ RMRs accessing $x \triangleright B$ in $H'$, as wanted. 

**Theorem 8.17.** The implementation $I'_{EW}$ satisfies Specifications 6.1 and 6.2 under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the CC model. Finally, $I'_{EW}$ satisfies the locality property in the write-through and write-back CC models (Definitions 7.2 and 7.3).

**Proof.** Specification 6.1 (linearizability) under Condition 6.4 follows directly from Lemma 8.11 and Lemma 8.12. Specification 6.2 (termination) under Condition 6.4 follows from Lemma 8.13. $O(1)$ RMR complexity follows from Lemma 8.14, Lemma 8.15, as well as the fact that during any operation execution on the target object, a process accesses at most two blocks and accesses the field $B$ at most once per block per operation execution. Locality follows from Lemma 8.16.

### 8.2.2 DSM Model

In the DSM model, we construct a locally-accessible ECAS implementation $I_{EW-DSM}$ from $I_{EW}$ using a transformation analogous to the one presented in Section 7.3. That is, we designate a process $p_{\text{special}}$, and make the block manager, as well as the block fields $B$ and $\text{writer}$ locally-accessible to $p_{\text{special}}$. For the base object $B$, we achieve locality using the ECAS implementation from Section 7.3.

**Theorem 8.18.** The implementation $I_{EW-DSM}$ satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the DSM model. Finally, $I_{EW-DSM}$ satisfies the locality property in the DSM model with respect to the designated process $p_{\text{special}}$. 

Proof. This theorem is analogous to Theorem 7.35 in Section 7.3, and its proof is also analogous. As in Section 7.3, we must also show that the block manager is accessed according to Condition 7.18 in histories of $I_{EW-DSM}$, since the locally accessible block manager implementation depends on this. To that end, the analog of Lemma 7.34 follows from a proof analogous to the one given in Section 7.3 except that for Condition 7.18 (a) we consider the structure of the access procedure for operation type Write instead of ECAS.

8.3 Conclusion

In this chapter we showed how to make writable our implementation of ECAS (and hence our implementations of CAS and LL/SC). In Section 8.2 we showed how to do this in a manner that preserves the locality properties defined in Chapter 7. Using our results from Chapters 6–8 we are now ready to establish our principal result (3) from Chapter 1. We do so next in Chapter 9.
Chapter 9

Simulation of Algorithms Based on Comparison Primitives and LL/SC Using Reads and Writes Only

In this Chapter we establish our principal result (3) from Chapter 1, which states: “any CC or DSM shared memory algorithm using read, write, comparison primitives and LL/SC can be simulated by an algorithm that uses only read and write operations, with only a constant-factor increase in the RMR complexity, while preserving other important properties.” We explained at the end of Chapter 6 why our $O(1)$-RMR implementations of CAS and LL/SC from that Chapter are not sufficient for this purpose. (The same reasoning applies to other comparison primitives, which we discuss shortly.) Our observations there motivated the material in Chapters 7 and 8. Next we will show how to simulate any comparison primitive using CAS, which we now know how to implement using reads and writes, and then finally prove our principal result (3).

9.1 Simulation of Comparison Primitives Using CAS

Anderson and Kim define comparison primitives [2] as a class of synchronization primitives that includes CAS and TAS (test-and-set). A generic comparison primitive can be thought of as an object type supporting an operation type `compare_and_fg` (in addition to `Read` and `Write`), which is parametrized by functions $f$ and $g$, and corresponds to the atomic execution of the pseudo-code shown in Figure 9.1. As an example, this generic definition can be instantiated to CAS by defining $f(cmp, new) \equiv new$ and
Chapter 9. Simulation of Algorithms

\[ g(old, cmp, new) \equiv old. \]

Function `compare_and_fg(cmp, new)

\[ \begin{align*}
302 & \quad old := S \\
303 & \quad \text{if } old = cmp \text{ then } S := f(cmp,new) \\
304 & \quad \text{return } g(old, cmp, new)
\end{align*} \]

Figure 9.1: Definition of a comparison primitive. (The current state is denoted by \( S \).)

Any comparison primitive can be implemented by using a CAS implementation (e.g., the one described in Chapters 6, 7 and 8 as a black box. Specifically, we record the state of the target object using a CAS object, and access the state using CAS operations (as well as `Read` and `Write`). To perform a `compare_and_fg` operation execution, we execute the pseudo-code above with lines 302–303 replaced by the following statement:

\[ old := \text{CAS}(cmp, f(cmp,new)) \]

Specifications 6.1 and 6.2 (linearizability and termination) follow easily from the structure of this very simple implementation. The locality property defined in Chapter 2 also holds if a locally-accessible CAS implementation is used, such as the one described in Chapters 7 and 8.2. This follows by a straightforward proof similar to the one given for Theorem 7.4 in Chapter 7. (In this case we treat a `compare_and_fg` operation execution as read-like when the argument `cmp` is different from the prior state, and write-like otherwise.)

9.2 Principal Result

The results described up to this point culminate in Theorem 9.1 below, which states precisely our principal result (3) from Chapter 1. The theorem assumes that an algorithm \( \mathcal{A} \) does not access the same shared object using both a comparison primitive and LL/SC. This is a reasonable assumption because multiprocessors typically provide either one type of primitive or the other. The assumption coincides with Condition 6.4 in Chapter 6, which we need for technical reasons to ensure the linearizability of our implementations.

Informally, Theorem 9.1 states that any algorithm \( \mathcal{A} \) can be simulated by an algorithm \( \mathcal{A}' \) that does not rely on comparison primitives or LL/SC, with only a constant-factor increase in RMR complexity. Of course this claim is satisfied trivially unless the transformation preserves fundamental correctness properties. It is difficult to state precisely
what properties are or are not preserved by our transformation from $\mathcal{A}$ to $\mathcal{A}'$, and so we characterize only a subset of the properties that are preserved (see properties (c)–(d) of $\mathcal{A}'$ stated in Theorem 9.1). This subset includes any safety property that can be defined (solely) in terms of the state of register objects used in $\mathcal{A}$ (which also appear in $\mathcal{A}'$), as well as common liveness properties.

As an example of how Theorem 9.1 can be applied, suppose that $\mathcal{A}$ is a mutual exclusion (ME) algorithm that uses reads, writes and LL/SC. Then $\mathcal{A}'$ is also an ME algorithm and uses reads and writes only. To see why the ME property is preserved, note that $\mathcal{A}$ (and hence $\mathcal{A}'$) can be instrumented so that any violation of ME in a history $H'$ can be inferred from $H|R$ or $H'|R$, where $R$ is the set of register objects used in $\mathcal{A}$. To that end, we introduce a global shared register $r$ that is first read and then written in each execution of the critical section. If the critical section is not executed in mutual exclusion in $\mathcal{A}'$, then in some history $H'$ of $\mathcal{A}'$ the operations on $r$ by two distinct processes inside the CS are interleaved in such a way that two consecutive operations on $r$ in $H'$ are reads. The same holds in $H$ by property (c) above, which implies that the critical section is not always executed in mutual exclusion in $\mathcal{A}$ either.

Theorem 9.1 also implies that if $\mathcal{A}$ is a ME algorithm that satisfies deadlock or starvation freedom, then $\mathcal{A}'$ satisfies the same liveness property. For starvation freedom, this follows directly from property (d) of $\mathcal{A}'$ in Theorem 9.1. For deadlock freedom, this follows from properties (c)–(d), where property (c) is used to show that if the critical section is executed finitely many times in $H'$, the same holds for $H$. (The number of times the CS is executed can be deduced by instrumenting $\mathcal{A}$ and $\mathcal{A}'$ as in the analysis of ME.)

**Theorem 9.1.** For any of the three shared memory models discussed in this thesis (i.e., write-through CC, write-back CC or DSM), and for any algorithm $\mathcal{A}$ that uses atomic read/write registers, as well as (readable/writable) shared objects that support either comparison primitives or LL/SC, let $\mathcal{A}'$ be the algorithm constructed from $\mathcal{A}$ as follows:

1. Simulate every application of a comparison primitives using the $\text{CAS}$ operation type, as described in Section 9.1.

2. Replace any shared object accessed using $\text{CAS}$ or LL/SC with our readable/writable, locally-accessible $O(1)$-RMR implementation. (See Section 8.2, which builds on Chapters 6–8.)

Then $\mathcal{A}'$ has the following correctness properties:
(a) it uses read/write registers only; and
(b) it has the same RMR complexity as $A$, up to a constant factor, when executed in
the particular shared memory model under consideration; and
(c) letting $R$ denote the set of objects accessed using only reads and writes in $A$, for
any history $H'$ of $A'$ there is a history $H$ of $A$ where $H'|R = H|R$; and
(d) for any fair history $H'$ of $A'$, there is a fair history $H$ of $A$ where the same processes
are active, and each active process either terminates in both histories, or does not
terminate in either history.

Proof.
Property (a): The transformation from $A$ to $A'$ replaces any objects other than
read/write registers with our implementations, which themselves use only reads and
writes, provided the base objects and subroutines are chosen as stated.

Property (b): Recall that the locality properties imply the RMR-preservation prop-
erty (see Chapter 7). Step 1 in the transformation from $A$ to $A'$ introduces at most
a constant-factor increase in RMR complexity, since an $O(1)$-RMR implementation of
CAS with the locality property is used to simulate other comparison primitives. Simi-
larly, step 2 introduces at most a constant-factor increase in RMR complexity because
implementations with locality properties are used. Thus, $A'$ has the same RMR com-
plexity as $A$, up to a constant factor, when both algorithms are executed on the same
multiprocessor (of one of the types considered in this thesis).

Properties (c) and (d): Suppose we are given a history $H'$ of $A'$. Let $H$ be the history
obtained by “reversing” the two steps in the transformation used to obtain $A'$ from $A$
as follows. First, we replace calls to access procedures in our implementations of CAS
and LL/SC (each of which consists of one or more atomic steps) by operation executions
(i.e., invocation and response pairs) on the corresponding objects of type CAS or LL/SC
used in $A$. Next, determine the linearization of these operation executions as defined
in our proof of linearizability, and replace the operation executions with atomic steps,
where each atomic step is scheduled between the corresponding invocation and response
in $H'$ in a manner consistent with the linearization order. For operation executions that
are pending, we record an atomic step if the corresponding operation execution on the
target object has taken effect (i.e., appears in the linearization), and we discard the
operation execution otherwise. Finally, we replace certain CAS operation executions with
applications of other synchronization primitives, as needed to “reverse” step 1 of the
transformation.

Since the implementations used in both steps of the transformation from \( \mathcal{A} \) to \( \mathcal{A}' \) satisfy Specification 6.1 (linearizability), it follows that the history \( H \) is a history of \( \mathcal{A} \). Furthermore, \( H|R = H'|R \) follows by our construction of \( H \) from \( H' \), and so part (c) holds. For part (d), note that if \( H \) is fair then a process is active in \( H \) if and only if it is active in \( H' \), and furthermore since our implementations satisfy Specification 6.2 (termination), the following property also holds: \( p \) takes infinitely many steps in \( H' \) only if it does so in \( H \).

Having stated Theorem 9.1 and given its proof, we now give examples of properties that are not preserved by the transformation from \( \mathcal{A} \) to \( \mathcal{A}' \). First, if \( \mathcal{A} \) is an FCFS ME algorithm, then \( \mathcal{A}' \) is not necessarily an FCFS algorithm. This is because \( \mathcal{A} \) has a section of code called the *doorway*, which by definition terminates in \( O(1) \) steps. The analog of this doorway also appears in \( \mathcal{A}' \), but it does not necessarily terminate in \( O(1) \) steps. For example, if a comparison primitive is applied in the doorway of \( \mathcal{A} \), then by replacing objects that are accessed in \( \mathcal{A} \) using comparison primitives or LL/SC with our simulations of these objects, we introduce busy-wait loops. (These busy-wait loops appear in pseudolocks and the underlying name consensus algorithm.) Thus, our transformation does not in general preserve the FCFS property. Wait-freedom is another property that is not preserved, for analogous reasons.

### 9.3 Conclusion

At this point we have proved our principal results (1)–(3) from Chapter 1. Along the way we have presented several algorithms and shared object implementations. One of the greatest challenges so far has been to design locally-accessible implementations (see Chapter 7 and Section 8.2) and so it is natural to ask whether locality can be achieved using a universal construction. For the CC model we leave the question open. For the DSM model we present a universal construction in Chapter 10. We then show how to bound the space complexity of all our implementations (from Chapters 6–10) in Chapters 11–12.
Chapter 10

Locally-Accessible Shared Objects in the DSM Model

In this chapter we present a linearizable implementation $I_U = (\tau, \mathcal{P}, \mathcal{B}, \mathcal{H})$ of any type $\tau$ that uses only reads and writes (or shared objects, such as the block manager, that can be implemented using reads and writes). In the DSM model this implementation satisfies the locality property with respect to a designated process $p_{\text{special}}$. In addition, $I_U$ has $O(1)$ RMR complexity per operation execution in the DSM model under a certain condition discussed shortly (see Condition 10.2 below). The ID of $p_{\text{special}}$ is fixed as in Chapter 7 rather than being determined at runtime. The rationale for this is as in Chapter 7—we intend to use the implemented object to simulate its hardware counterpart, and so we would like the implemented object to behave like its hardware counterpart (which is local to a statically-assigned processor) in terms of RMR complexity.

The universal technique underlying $I_U$ is motivated by the difficulty of constructing shared objects that satisfy the locality property in the DSM model. In earlier chapters we showed how to do this for several shared object types including LL/SC and comparison primitives (see Chapters 7 and 9), and even in those special cases our algorithms are somewhat complex. Recall for example our use of sequence numbers in Section 7.3, and the problem of these numbers growing without bound. Naturally the technical challenges become even greater when we attempt to create implementations that satisfy the locality property in the DSM for other shared object types. The source of difficulty is twofold. First, since the locality of a variable is determined statically, it is more difficult in general to write programs that are efficient in terms of RMRs in the DSM model, compared to the more common write-back CC model. Second, in the special case of implementations
that satisfy the locality property in the DSM model, the algorithms for the designated process $p_{special}$ and for non-special processes tend to be very different conceptually, which means that two sets of pseudo-code are required for each operation type, and the analysis becomes correspondingly longer. This is in contrast to the CC model where typically the access procedure can be expressed using common pseudo-code for all processes.

To illustrate the second point above, consider a history of an implementation where the designated process $p_{special}$ and one non-special process $q$ access an implementation that satisfies the locality property in the DSM model and has bounded RMR complexity. In this case $q$ cannot busy-wait for a signal from $p_{special}$ because such a signal would require an RMR by $p_{special}$ (or else $q$’s algorithm incurs unbounded RMRs, or $q$’s algorithm does not terminate.) Thus, in the two-process case, the access procedures for $q$ must be wait-free (see Chapter 1), whereas the access procedures for $p_{special}$ necessarily cannot be (for every operation type) except when the target type occupies level one in Herlihy’s wait-free hierarchy [14] – an uninteresting scenario because it excludes common shared object types such as queues, stacks, counters, comparison primitives and LL/SC.

The above example shows that in constructing implementations that satisfy the locality property in the DSM model (in addition to bounded RMR complexity and termination), we face some of the same challenges encountered in wait-free synchronization. Not surprisingly then, similar algorithmic techniques can be applied in both scenarios. In fact, it is possible to use a wait-free universal construction as a building block for $I_U$, which is what we do in this chapter (albeit in an unusual manner).

As mentioned earlier, the correctness properties of $I_U$ (linearizability with respect to an arbitrary type $\tau$, termination, $O(1)$ RMR complexity, and locality to a designated process $p_{special}$) are contingent on a certain condition. Such a condition cannot be avoided entirely because known impossibility results (see Chapter 1) preclude implementations that achieve the first three correctness properties listed above for certain shared object types, namely those than can be used (in conjunction with reads and writes) to solve mutual exclusion with fewer than $\Theta(\log N)$ RMRs per process in the worst case. The condition underlying $I_U$ is that operation executions of certain types must not be applied concurrently. To characterize these operation types, we first introduce the following definition:

**Definition 10.1.** For any shared object type $\tau = (S, s_{init}, O, R, \delta)$ and any operation type $ot \in O$, we say that $ot$ is purely read-like if and only if for every state $s \in S$, $\delta(s, ot)$
is a set of tuples of the form \((s, \ldots)\). We also say that an operation (i.e., an atomic step or operation execution) is purely read-like if its operation type is purely read-like.

Informally, Definition 10.1 states that an operation of type \(ot\) does not change the state of the shared object, regardless of the state in which it is applied. For example, \texttt{Read} is purely read-like but \texttt{CAS} is not. (Note that the concept of “read-like” and “write-like” defined earlier in Chapter 7 referred to operation executions, while Definition 10.1 refers to operation types as well.) Now the condition needed for \(I_U\) to have \(O(1)\) RMR complexity is as follows:

**Condition 10.2.** Non-special processes concurrently apply \(O(1)\) operation executions of types that are not purely read-like (see Definition 10.1).

As hinted earlier, we obtain \(I_U\) through a universal construction technique that uses a universal wait-free construction as a building block. To simplify exposition, we present the construction technique in several stages, building up successively stronger implementations denoted \(I^1_U\), \(I^2_U\), \(I^3_U\) and \(I^4_U\). As we show in Section 10.4, \(I^4_U\) satisfies all the properties defined above for \(I_U\).

Before describing our implementations, we point out the subtle issue of register size. Our implementation techniques require \(O(\log N)\)-bit registers except in the special case when a register is used to record a state of the target type \(\tau\). Thus, our implementations can be obtained using \(\Theta(\log N)\)-bit registers provided that the target type \(\tau\) satisfies the following:

**Condition 10.3.** Any state of type \(\tau\) can be represented using \(O(\log N)\) bits.

### 10.1 Implementation \(I^1_U\) for Two Processes

We begin by constructing an implementation \(I^1_U\) that satisfies all the desired correctness properties but only in the special case when \(N\) (the total number of processes) is two. We construct implementation \(I^1_U\) using a slightly modified version of the universal wait-free construction of Jayanti and Toueg [13]. Recall that this construction is similar to Herlihy’s [14], except that it uses registers of bounded size. Both constructions use read/write registers as well as shared objects of type \texttt{consensus}, which is denoted \(\tau_{\text{CON}}\) subsequently and is defined as follows: The set of states consists of values from some domain \(U\), and a special value \(\bot \not\in U\). The initial state is \(\bot\). The operation type defined
are \texttt{Decide}(v) for each \( v \in U \), and \texttt{Reset}(). The transition relation is as follows: a \texttt{Decide}(v) operation applied in state \( \bot \) changes the state to \( v \) and returns \( v \), otherwise it returns the state without changing it; a \texttt{Reset}() operation changes the state to \( \bot \) and returns \( OK \).

We now give a high-level overview of the Jayanti-Toueg wait-free universal construction for the unfamiliar reader. In this construction (as in Herlihy’s) the state of the target object is recorded in data structures called \textit{cells}. As processes apply operation executions on the target object, the corresponding sequence of states is recorded in a chain of cells tagged with consecutive sequence numbers. Thus, each cell records not only a state of the target type, but also a pointer to the next cell in the chain and a sequence number. At the beginning of any operation execution, the calling process \( p \) allocates a new cell \( C_p \), writes the type of its operation execution in \( C_p \), and writes the address of \( C_p \) at index \( p \) of a shared array \textit{announce}[1..N]. (This announces \( p \)'s intentions to other processes, which may help \( p \) apply its operation execution as described shortly.) Then, \( p \) identifies a cell near the end of the chain by scanning another array, \textit{head}[1..N], where element \textit{head}[q] is used by process \( q \) to record a pointer to the last cell \( q \) has threaded onto the chain. By comparing the sequence numbers of the cells recorded in \textit{head}[1..N], \( p \) finds the cell that is closest to the end of the chain. From that cell, \( p \) follows the chain to locate the end.

It is possible that \( p \) never finds the end of the chain because new cells are threaded repeatedly, but in that case after following roughly \( N \) cells \( p \) knows that some other processes \( q \) has threaded \( C_p \) on behalf of \( p \). (This is the helping mechanism referred to earlier, and explained further below.) In that case \( p \) can complete its operation execution after computing the correct response from information written to \( C_p \) by \( q \). If \( p \) does find the last cell \( C_l \) in the chain, it tries to thread another cell onto \( C_l \). In order to ensure wait-freedom, \( p \) must not only try to thread its own cell, but must also try to “help” other processes by threading cells recorded by them in \textit{announce}[1..N]. To determine the next cell \( C_h \) to be threaded, \( p \) uses the sequence number in \( C_l \) to compute an array index \( i \) and examines \textit{announce}[i]. If it finds a pointer to cell there, then \( C_h \) becomes this cell, otherwise \( C_h \) becomes \( C_p \). To handle the race condition where multiple processes attempt to thread cells onto \( C_l \), a shared object of type \( \tau_{CON} \) is used. Specifically, \( p \) applies \texttt{Decide}(C_h) to the consensus object in \( C_l \). Process \( p \) then reads the response \( C_a \) of this \texttt{Decide} operation to determine which cell was actually threaded onto \( C_l \) (i.e., \( C_h \) or another cell threaded by a competing process), and then updates three fields in
Chapter 10. Locally-Accessible Shared Objects in the DSM Model 129

\( C_a \): the sequence number, the new state represented by the cell, and the response of the operation represented by \( C_a \). The sequence number is one more than in \( C_l \). The new state and response are determined by applying the transition relation of the target type to the previous state recorded in \( C_l \) and the operation type recorded in \( C_a \). Next, if \( p \) has succeeded in threading \( C_h \), it assigns \( head[p] = C_h \). Finally, \( p \) checks whether \( C_p \) has been threaded, and if so then \( p \) computes the response of its operation execution from the state recorded in \( C_p \), and returns. Otherwise, \( p \) selects another cell \( C_h \) and tries to thread that cell onto \( C_a \) using the same algorithm as described above. (Since the sequence number of \( C_a \) is one higher than \( C_l \)'s, this ensures that \( p \) tries to help a different process in each iteration, including itself eventually.)

The Jayanti-Toueg construction also provides a mechanism to recycle cells that have been threaded onto the chain and are no longer needed. The details of this mechanism are somewhat complex because it is difficult to determine when it is safe to recycle a particular cell. However, once such a cell has been identified, recycling it is simple – the registers in the cell are reset to their initial values, and the consensus object in the cell is reset using a call to \( \text{Reset}() \).

The Jayanti-Toueg construction for two processes yields a linearizable \( O(1) \)-RMR implementation of type \( \tau \) when \( N = 2 \), but such an implementation is not sufficient for our purposes in this section because of the following problems: (1) it does not satisfy the locality property in the DSM model (with respect to either of the two processes), and (2) base objects of type \( \tau_{CON} \) are required in addition to read/write registers. Still, this implementation is a useful starting point for \( I_1U \). To obtain \( I_1U \), we transform the two-process instance of the Jayanti-Toueg construction for type \( \tau \) as follows: To achieve the locality property in the DSM model with respect to some designated process \( p_{special} \), it suffices to make all the base objects local to \( p_{special} \). (This includes all the cells used in the Jayanti-Toueg construction.) To solve the second problem, it suffices to simulate consensus objects using read/write registers while preserving \( O(1) \)-RMR complexity and locality with respect to \( p_{special} \).

A simple implementation of consensus from read/write registers can be obtained using the \( O(1) \)-RMR two-process leader election algorithm local to \( p_{special} \) described in [11]. (See Specification 3.9 and Figure 3.2 in Section 3.3.) To simplify matters, we will take advantage of two observations: in the Jayanti-Toueg construction (1) no operation execution is applied concurrently with a \( \text{Reset}() \) and (2) each process calls \( \text{Decide}(...) \) at most once between two \( \text{Reset}() \) operations. Our implementation of consensus for
Chapter 10. Locally-Accessible Shared Objects in the DSM Model

130

\( p^1_{\text{special}} \) and a non-special process \( q \) uses one instance of the LE algorithm (see Figure 3.2), and two additional shared variables: registers \( \text{val}_{\text{special}} \) and \( \text{val}_{\text{other}} \), both local to \( p^1_{\text{special}} \) and initially \( \bot \). To apply \( \text{Decide}(v) \), \( p^1_{\text{special}} \) writes \( v \) to \( \text{val}_{\text{special}} \) and executes the LE algorithm. If it wins, it returns \( v \), otherwise it reads \( \text{val}_{\text{other}} \) and returns the value read. Similarly, to apply \( \text{Decide}(v) \) process \( q \) writes \( v \) to \( \text{val}_{\text{other}} \) and executes the LE algorithm. If it wins, it returns \( v \), otherwise it reads \( \text{val}_{\text{special}} \) and returns the value read. To apply \( \text{Reset}() \), a process simply resets each register used in the LE algorithm, as well as \( \text{val}_{\text{special}} \) and \( \text{val}_{\text{other}} \), to their initial values.

Given our simplifying observations, it is straightforward to show that the above implementation of consensus satisfies Specifications 6.1 (linearizability) and 6.2 (termination). \( O(1) \) RMR complexity of \( \text{Decide}(v) \) follows from the RMR complexity of the LE algorithm. \( O(1) \) RMR complexity of \( \text{Reset}() \) follows from the fact that only two variables are used in the LE algorithm and both are local to \( p^1_{\text{special}} \) (see [11]). Similarly, locality to \( p^1_{\text{special}} \) follows from the locality of the LE algorithm to \( p^1_{\text{special}} \) (Specification 3.9) and the fact that all the shared variables accessed during a \( \text{Reset}() \) are local to \( p^1_{\text{special}} \).

Theorem 10.4. The implementation \( I_U^1 \) of type \( \tau \) for \( N = 2 \) processes described in this section satisfies Specifications 6.1 (linearizability) and 6.2 (termination). Furthermore, each operation execution on the target object incurs \( O(1) \) RMRs in the DSM model. Finally, \( I_U^1 \) satisfies the locality property in the DSM model with respect to the designated process \( p^1_{\text{special}} \).

Proof. Recall that \( I_U^1 \) is obtained from a two-process instance of the Jayanti-Toueg construction for type \( \tau \) by

(a) making all register base objects local to \( p^1_{\text{special}} \), and

(b) simulating objects of consensus type using a \( O(1) \)-RMR linearizable implementation that satisfies the locality property in the DSM model with respect to \( p^1_{\text{special}} \).

Specifications 6.1 and 6.2 follow from corresponding properties of the Jayanti-Toueg construction and the implementation used to simulate base objects of type \( \tau_{\text{CON}} \). \( O(1) \) RMR complexity follows similarly since we instantiate the Jayanti-Toueg construction in the two-process case and since the implementation of consensus has \( O(1) \) RMR complexity. (For \( N \) processes, the RMR complexity of the Jayanti-Toueg construction is \( O(N^2) \).) Locality follows from the fact that all the base objects used in the Jayanti-Toueg construction are declared local to \( p^1_{\text{special}} \), and from the locality property of the
implementation of consensus.

\[ \square \]

## 10.2 Implementation $I_U^2$ for $N$ Processes with Restricted Concurrency

Next, we show how to transform $I_U^1$ into an implementation $I_U^2$ of type $\tau$ for $N$ processes that satisfies all the desired correctness properties under the following restriction:

**Condition 10.5.** Non-special processes do not apply operation executions concurrently.

Implementation $I_U^2$ is similar to $I_U^1$ in the sense that it may be accessed by at most two processes at a time, namely a designated process $p^2_{special}$ and at most one non-special process. It is strictly stronger than $I_U^1$, however, because the ID of the other process need not be fixed beforehand, and moreover multiple non-special processes may access $I_U^2$ in the same history provided they do so non-concurrently.

We obtain $I_U^2$ from $I_U^1$ by using the same base objects and making some subtle but important changes to the access procedures. Recall that in the Jayanti-Toueg construction, process IDs are used within the algorithm in two cases: (1) for wait-freedom, processes access shared arrays $\text{announce}[1..N]$ and $\text{head}[1..N]$ using their own ID as the array index; and (2) for linearizability with respect to type $\tau$, a process $p$ records in a cell the operation type of the operation execution it is applying, and its ID is embedded in that operation type. Our transformation from $I_U^1$ to $I_U^2$ applies a translation, when appropriate, from the set of process IDs in $I_U^1$ (i.e., $\{1, 2, \ldots, N\}$) to the set of process IDs in $I_U^2$ (i.e., $\{1, 2\}$). Let $p^2_{\text{special}}$ denote the ID of the process to which $I_U^2$ will be local, and ignore for now the locality of $I_U^1$. In the first case above, $p^2_{\text{special}}$ uses the ID 1 when indexing an array, and any non-special process $q$ uses the ID 2. In the second case, each process simply uses its own ID.

Another subtle point that must be addressed in our transformation is how to simulate consensus objects from reads and writes. For $I_U^2$ we can use the same technique as for $I_U^1$, based on two-process leader election. That is, process $p^2_{\text{special}}$ executes the same access procedure as described for $p^1_{\text{special}}$ before, and any non-special process executes the access procedure described for $q$. (In the LE algorithm executed inside this access procedure, the ID of the caller is never used for control flow and is never written to a shared variable, which is why the ID of the non-special process need not be fixed beforehand.
See Figure 3.2.)

Our description of $I_U^2$ up to this point deals with linearizability and termination. Now consider $O(1)$ RMR complexity and locality to $p_{special}^2$. To satisfy these properties, we follow the same approach as for $I_U^1$. That is, we make all the base objects local to $p_{special}^2$, including the consensus objects.

**Theorem 10.6.** The implementation $I_U^2$ of type $\tau$ for $N$ processes described in this section satisfies Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 10.5. Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the DSM model. Finally, $I_U^1$ satisfies the locality property in the DSM model with respect to the designated process $p_{special}^2$.

**Proof.** Recall that $I_U^2$ is obtained through a transformation of $I_U^1$ that

(a) translates process ID from \{1, 2, ..., N\} to \{1, 2\} in some cases, and

(b) makes all the base objects local to $p_{special}^2$.

First note that our modification to the Jayanti-Toueg construction (point (a) above) preserves the linearizability and termination properties under Condition 10.5. Thus, Specifications 6.1 and 6.2 for $I_U^2$ follow from the corresponding properties of the (modified) Jayanti-Toueg construction and the implementation used to simulate base objects of type $\tau_{CON}$. $O(1)$ RMR complexity follows similarly since the modified Jayanti-Toueg construction has the same RMR complexity as in the two-process case. Locality follows as in the proof of Theorem 10.6 because we make all the base objects local to $p_{special}^2$. □

### 10.3 Implementation $I_U^3$ for $N$ Processes

Implementation $I_U^2$ is weak in the sense that it satisfies Specifications 6.1 and 6.2 (linearizability and termination) only under Condition 10.5 (see Theorem 10.6). In this section we show how to obtain a stronger implementation $I_U^3$ that satisfies Specifications 6.1 and 6.2 unconditionally, and similarly for locality with respect to a designated process $p_{special}^3$. The RMR complexity of $I_U^3$ is $O(1)$ provided that the following condition holds:

**Condition 10.7.** Non-special processes apply $O(1)$ operation executions concurrently at any point.

Note that Condition 10.7 is strictly weaker than Condition 10.5 from Section 10.2.
Our construction of $I_3^3$ uses a single base object $B$ of type $\tau$ instantiated using $I_2^3$ and local to $p_3^{\text{special}}$. To satisfy Condition 10.5, we simply serialize operation executions by non-special processes using a mutual exclusion algorithm (i.e., a mutex). To satisfy $O(1)$ RMR complexity, the mutex algorithm must be chosen carefully because any mutex based on read/write registers has worst-case RMR complexity $\Omega(\log N)$ (see Chapter 1). Since at most $O(1)$ non-special processes apply operation executions concurrently under Condition 10.7, it suffices to use Kim and Anderson’s algorithm [17], which has $O(1)$ RMR complexity in that special case.

**Theorem 10.8.** The implementation $I_3^3$ of type $\tau$ for $N$ processes described in this section satisfies Specifications 6.1 (linearizability) and 6.2 (termination). Furthermore, each operation execution on the target object incurs $O(1)$ RMRs in the DSM model under Condition 10.7. Finally, $I_3^3$ satisfies the locality property in the DSM model with respect to the designated process $p_3^{\text{special}}$.

**Proof.** Note that our use of a mutex ensures that $B$, which is obtained using $I_2^3$, is accessed according to Condition 10.5. Thus, linearizability follows from Theorem 10.6 and the structure of the access procedures in $I_3^3$, where for each operation execution a process simply applies the corresponding operation on $B$ and returns the response. (This structure is implicit in our informal description above.) Similarly, termination follows from Theorem 10.6 and the correctness properties of the mutex. Locality again follows from Theorem 10.6 since the access procedures for $p_3^{\text{special}}$ make only one access to $B$ and do not access the mutex. \(\square\)

### 10.4 Implementation $I_4^4$ for $N$ Processes

Implementation $I_3^3$ is stronger than $I_2^3$, but it is still weak in the sense that it satisfies $O(1)$ RMR complexity in the DSM model only under Condition 10.7. In this section we show how to obtain a stronger implementation $I_4^4$ that has the same properties as $I_3^3$ (see Theorem 10.8), except that $O(1)$ RMR complexity holds under Condition 10.2, which is weaker than Condition 10.7. If Condition 10.2 does not hold, $I_4^4$ has the same worst-case RMR complexity as $I_3^3$ when Condition 10.7 does not hold: $O(\log N)$ RMRs.

Our construction of $I_4^4$ begins with a single base object $B$ of type $\tau$ instantiated using $I_3^3$. The high-level idea is to map operation executions in $I_4^4$ to atomic steps on $B$ in such a way that $B$ records the state of the target object. To that end, the designated
process \( p_{\text{special}} \) applies each operation execution by performing the corresponding operation on \( B \), and returning the response. For non-special processes, this does not work because Condition 10.7 restricts the manner in which they may access \( B \). To satisfy Condition 10.7, the naive solution would be to serialize operations on \( B \) using a mutex, which is too costly in terms of RMRs. This is because up to \( N \) processes may apply purely read-like operation executions concurrently in \( I_U^4 \) under Condition 10.2, and so the overhead introduced by the mutex would be \( \Omega(\log N) \) (see Chapter 1) since the mutex must be based on read/write registers only. Instead, we use the specialized synchronization techniques developed in earlier chapters, which are more RMR-efficient. To that end, we take advantage of the fact that purely read-like operation executions never change the state of the target object. Thus, we can apply entire groups of such operation executions by electing a leader that accesses \( B \) and shares the response with the others. This is somewhat similar to the way concurrent ECAS operation executions are synchronized in Chapter 6.

The access procedures for implementation \( I_U^4 \) are presented in Figure 10.1. The implementation uses an instance \( M \) of the block manager described in Chapter 5. (Note that \( p_{\text{special}} \) does not access the block manager object, and so the version from Chapter 5 suffices; we do not need the locally-accessible one described in Section 7.3.) Each block contains a register \( val \) and a pseudo-lock which is used to synchronize groups of concurrent read-like operation executions. We present the implementation of a generic purely read-like operation type \( \text{Op-R} \), and a generic operation type \( \text{Op-W} \) of the other variety (i.e., non-purely-read-like). These serve as templates for implementing the actual operation types defined in the shared object type \( \tau \) under consideration.

The access procedures for \( \text{Op-R} \) and \( \text{Op-W} \) by \( p_{\text{special}} \) simply apply the corresponding operation to \( B \) and return the response (see lines 306 and 312). The access procedure for \( \text{Op-W} \) by a non-special process is the same as for \( p_{\text{special}} \). Finally, the access procedure for \( \text{Op-R} \) by a non-special process relies on a subroutine \( \text{ReadlikeHelper()} \) that returns the state of \( B \), which is then used to compute the response of \( \text{Op-R} \) at line 310. To ensure linearizability, \( \text{ReadlikeHelper()} \) must be called twice (see lines 308–309): this way, the value returned is guaranteed to be the current state of \( B \) at some point during the corresponding execution of \( \text{Op-R} \) (see Lemma 10.12).

The subroutine \( \text{ReadlikeHelper()} \) is called only by non-special processes, and works as follows. First, the caller, say \( q \), determines the current block \( d \) at line 313, and tries to acquire the pseudo-lock in block \( d \) at line 314. (This pseudo-lock is different
Chapter 10. Locally-Accessible Shared Objects in the DSM Model

from the one used internally by the block manager.) If \( q \) acquires the pseudo-lock, it determines the state of \( B \) at line 315, and writes this state to \( d \mapsto val \) at line 316. We assume without loss of generality that \( B \) supports an operation type \( \text{GetState()} \) that returns the current state, and this is what \( q \) uses at line 315. (There is no loss of generality because \( B \) is obtained using the universal construction \( I^U_3 \), and so we are free to add operation types to \( \tau \). The more operation types we add, the harder it becomes to construct our implementations.) After line 316, process \( q \) tries to change the current block at lines 317–318, calls \( d \mapsto \text{Pseudo-Exit()} \) at line 319, and finally returns the response of its call to \( B.\text{GetState()} \) from line 315. If \( q \) fails to acquire the pseudo-lock at line 314, it simply reads \( d \mapsto val \) at line 321, and returns the value read.

Lemma 10.9. For any history \( H \) of \( I^U_3 \):

(a) For any block \( x \), the pseudo-lock in block \( x \) is accessed according to Condition 4.1.
(b) Every call to \( M.\text{chngCurBlock}(d, d') \) at line 318 succeeds, and makes current a block \( d' \) that has never been current before.
(c) A read of \( x \mapsto val \) at line 321 of \( \text{ReadlikeHelper()} \) returns the value written to \( x \mapsto val \) at line 316 by the unique process that acquired the pseudo-lock in block \( x \).

(The uniqueness of this process follows from part (a).)

Proof. Suppose, for contradiction, that the lemma does not hold. Let \( H \) be the shortest history at the end of which some process \( p \) is about to apply a step that may violate the lemma, and note that the lemma holds in \( H \). We proceed by cases on which part of the lemma \( p \)'s step may violate.

Case A: \( p \) is about to call \( x \mapsto \text{Pseudo-Enter()} \) or \( x \mapsto \text{Pseudo-Exit()} \) for some block \( x \) for the second time, violating part (a). Since \( |H| \) is minimal, it follows from the structure of the access procedure that \( p \) is about to make a call to \( x \mapsto \text{Pseudo-Enter()} \) at line 314.

Note that before its first call to \( x \mapsto \text{Pseudo-Enter()} \), \( p \) received \( x \) as the response of \( M.\text{getCurBlock()} \) at line 313, and so \( x \) was current at that point. If \( p \) acquired the pseudo-lock in block \( x \) when it first called \( x \mapsto \text{Pseudo-Enter()} \), then \( p \) subsequently called \( M.\text{chngCurBlock} \) at line 318, and so after that point \( x \) is never again current by part (b). But that contradicts \( p \) subsequently making another call to \( M.\text{getCurBlock} \) at line 313 with response \( x \) and then calling \( x \mapsto \text{Pseudo-Enter()} \) again at line 314.

If, on the other hand, \( p \) failed to acquire the pseudo-lock in block \( x \) when it first called \( x \mapsto \text{Pseudo-Enter()} \), then before \( p \) completed that call, by Specification 4.2 (c) and
part (a) for $H$ some other process $q$ applied $M.chngCurBlock$ at line 318 and then made a call to $x \triangleright Pseudo-Exit()$ at line 319. Once again block $x$ is never again current following this chngCurBlock operation by part (b), which contradicts $p$ eventually making another call to $x \triangleright Pseudo-Enter()$.

**Case B:** $p$ is about to make an unsuccessful call to $M.chngCurBlock(d, d')$ at line 318. In this case, some process $q$ must have previously applied a (successful) $M.chngCurBlock(d, ...)$, and so by the algorithm $q$ previously acquired the pseudo-lock in block $d$ at line 314. By the hypothesis of Case B, $p$ has also acquired the same pseudo-lock, which contradicts part (a) for $H$.

**Case C:** $p$ is about to make a successful call to $M.chngCurBlock(d, d')$ at line 318 where $d'$ is a block that was already current. Since any block that becomes current after the initial one is first returned by AllocBlock() at line 317 and then made current at line 318, the hypothesis of Case C contradicts Specification 5.1.

**Case D:** $p$ is about to read a value $v$ from $x \triangleright val$ at line 321 of ReadlikeHelper() other than one written by the unique process that acquired the pseudo-lock in block $x$. Having shown in Cases A–C that parts (a) and (b) of the lemma hold, we derive a contradiction by a proof similar to the one given for Lemma 5.5 (b) in Chapter 5.

**Lemma 10.10.** For any history $H$ of $I^4_U$, there is at most one process at lines 315–318 at any point in $H$.

**Proof.** First, we will show that if a process $q$ applies $M.getCurBlock()$ at line 313 in step $i$ of $H$ with response $x$ and then applies $M.chngCurBlock(x, ...)$ at line 318 in step $j$ of $H$ during the same call to ReadlikeHelper(), then $x$ is current between states $H[i]$ to $H[j]$. Suppose otherwise. Then some process $z$ applies a successful $M.chngCurBlock(x, ...)$ at line 318 in step $k$ of $H$, such that $i < k < j$. It follows from the algorithm that $z \neq q$ and that $z$ acquires the pseudo-lock in block $x$ before calling $M.chngCurBlock(x, ...)$. Since $q$ does the same, this means that two process acquire the pseudo-lock in block $x$, which contradicts Specification 4.2 (b) and Lemma 10.9 (a).

Now to prove the lemma, suppose for contradiction that distinct processes $p$ and $q$ are at lines 315–318 concurrently in some state $H[l]$, after applying $M.getCurBlock()$ at line 313 with response $x_p$ and $x_q$ (respectively). Then it follows that $x_p$ is current in state $H[l]$, as argued above, and similarly for $x_q$, which implies that $x_p = x_q$. Since we assume $p$ and $q$ are distinct, and since they acquire the pseudo-lock in block $x_p$ and $x_q$ (respectively) before state $H[l]$, this contradicts Specification 4.2 (b) and Lemma 10.9 (a).
Chapter 10. Locally-Accessible Shared Objects in the DSM Model

Declarations

**Shared variables:** (global)
- \( M \) – \( O(1)\)-RMR block manager from Chapter 5
- \( B \) – object of type \( \tau \) implemented using \( I^3_U \), local to \( p_{\text{special}} \), initialized to \( v_0 \) (i.e., the initial state of \( I^4_U \))

**Subroutines:** (global)
- \( \text{AllocBlock}() \) – \( O(1)\)-RMR block allocator

**Shared variables:** (per-block)
- \( \text{val} \) – register, uninitialized

**Subroutines:** (per-block)
- \( \text{Pseudo-Enter/Pseudo-Exit} \) – \( O(1)\)-RMR pseudo-lock from Chapter 4

**Private variables:** (per-process)
- \( \text{ret}, s \) – response of an operation type for type \( \tau \), uninitialized
- \( d, d' \) – block addresses, uninitialized

<table>
<thead>
<tr>
<th>Function ( \text{Op-R()} )</th>
<th>Function ( \text{ReadlikeHelper()} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>305 if ( \text{PID} = p_{\text{special}} ) then</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} return ( B.\text{Op-R}() )</td>
<td></td>
</tr>
<tr>
<td>306 else</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} ( \text{ReadlikeHelper()} )</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} ( s := \text{ReadlikeHelper()} )</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} return correct response for ( \text{Op-R()} ) applied in state ( s )</td>
<td></td>
</tr>
<tr>
<td>311 end</td>
<td></td>
</tr>
<tr>
<td>312 return ( B.\text{Op-W}() )</td>
<td></td>
</tr>
<tr>
<td>313 ( d := M.\text{getCurBlock}() )</td>
<td></td>
</tr>
<tr>
<td>314 if ( d \triangleright \text{Pseudo-Enter}() = \text{true} ) then</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} ( \text{ret} := B.\text{GetState}() )</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} write ( d \triangleright \text{val} := \text{ret} )</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} ( d' := \text{AllocBlock}() )</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} ( M.\text{chngCurBlock}(d,d') )</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} ( d \triangleright \text{Pseudo-Exit}() )</td>
<td></td>
</tr>
<tr>
<td>320 else</td>
<td></td>
</tr>
<tr>
<td>\hspace{1em} ( \text{ret} := \text{read}(d \triangleright \text{val}) )</td>
<td></td>
</tr>
<tr>
<td>321 end</td>
<td></td>
</tr>
<tr>
<td>322 return ( \text{ret} )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.1: Implementation \( I^4_U \) for the DSM model.

To prove linearizability, we now define for each history \( H \) of \( I^4_U \) a candidate linearization \( \bar{H} \) as follows. For each operation execution in \( H \) (complete or pending), we assign a “timestamp”, which is a tuple of the form \((t,q)\), where \( t \) is an integer and \( q \) is a process ID or zero.

**Definition 10.11.** The timestamp \( s \) for an arbitrary operation execution \( \text{Op} \) in \( H \), say by process \( p \), and its completion (where applicable), are defined as follows:
Chapter 10. Locally-Accessible Shared Objects in the DSM Model

Operation type Op-R():

(a) If Op is complete and p applies B.Op-R() at line 306 or B.GetState() at line 315 in step i of H, then s = (i, 0).

(b) Else if Op is complete and p does not access B during Op, then letting r denote the response of p’s call to ReadlikeHelper() at line 309, and letting i denote the earliest step (by any process) that occurs during Op and where B.GetState() is applied with response r, s = (i, p).

(We show that i is well-defined in Lemma 10.12 below.)

(c) Otherwise s is undefined.

Operation type Op-W():

(d) If p applies B.Op-W() at line 312 of Op-W in step i of H, then s = (i, 0).

(If Op is pending in H, its completion returns the response of p’s B.Op-W() in step i.)

(e) Otherwise s is undefined.


Proof. Consider p’s call to ReadlikeHelper() at line 309. If p itself executes line 315 during this call, then s is the value returned by B.GetState(), and so the lemma follows immediately. Otherwise, p reads $x \triangleright val$ at line 321 for some block x and so by Lemma 10.9 (c) it reads the value written to $x \triangleright val$ at line 316 by the unique process q that acquired the pseudo-lock in block x. In particular, by the algorithm this value is r and is the response of a B.GetState() by q at line 315 that occurs after q acquires the pseudo-lock in block x and before q writes $x \triangleright val$ at line 316.

To prove the lemma it suffices to show that q acquires the pseudo-lock in block x after p invokes Op, where x is the block accessed by p during its call to ReadlikeHelper() at line 309. To that end, note that x is not current when p calls M.getCurBlock() during its first call to ReadlikeHelper() (at line 308), otherwise p calls $x \triangleright Pseudo-Enter()$ in both its calls to ReadlikeHelper(), which contradicts Lemma 10.9 (a). Since x becomes current at most once in H by lines 317–318 of the algorithm and Specification 5.1 and
since only $p$’s second call to $M.getCurBlock()$ during $Op$ returns $x$, $x$ becomes current for the first time after $p$’s first call to $M.getCurBlock()$. Thus, when $q$ acquires the pseudo-lock in block $x$, $p$ has already applied its first $M.getCurBlock()$ during $Op$, as wanted.

To construct $\bar{H}$, we arrange operation executions for which timestamps are defined, in increasing order of timestamp. We prove the uniqueness of timestamps in Lemma 10.13, and define their order in Definition 10.14. (This is the same ordering as in Definition 7.27.)

**Lemma 10.13.** The timestamp of each operation execution in $H$ (for which the timestamp is defined) is unique.

*Proof.* This follows because if the timestamp $s$ of an operation execution $Op$ by process $p$ is $(i, q)$, then either $p$ applies step $i$ in $H$ during $Op$, or $Op$ is pending in state $H[i]$ and $q = p$. \(\Box\)

**Definition 10.14.** For timestamps $(t_1, q_1)$ and $(t_2, q_2)$, we say that $(t_1, q_1) < (t_2, q_2)$ if and only if $t_1 < t_2$, or, $t_1 = t_2$ and $q_1 < q_2$.

**Lemma 10.15.** $\bar{H}$ satisfies property (a) of linearizability (sequential completion).

*Proof.* This follows from the fact that any operation execution that is complete in $H$ has its timestamp defined. \(\Box\)

**Lemma 10.16.** $\bar{H}$ satisfies property (b) of linearizability (order preservation).

*Proof.* It follows from Definition 10.11 that if the timestamp of of some operation execution $Op$ in $H$ is $(i, \ldots)$, then $Op$ is pending in state $H[i]$. This and Definition 10.14 imply that if there are operation executions $Op_1$ and $Op_2$ in $H$ such that $Op_1$ precedes $Op_2$, then the timestamp of $Op_1$ is smaller than the timestamp of $Op_2$, as wanted. \(\Box\)

**Lemma 10.17.** Implementation $I_U^4$ satisfies property (c) of linearizability (conformity to type).

*Proof.* Let $H$ be any history of $I_U^4$. Since conformity to a type is a safety property it suffices to consider finite $H$.

We proceed as in the proof of Lemma 6.15, and consider only finite $H$. Let $k = |\bar{H}|$. Let $Op_i$, $s_i$, and $p_i$ denote the $i$’th operation execution in $\bar{H}$ (counting from 1), its timestamp, and the executing process. Let $\nu_i$ for $0 \leq i \leq k$ denote the state of a
correctly implemented object of type \( \tau \) after applying the first \( i \) operation executions in \( \bar{H} \). Define \( s_0 = (0, 0) \) and \( s_{k+1} = (\infty, 0) \). Let \((t_i, q_i)\) denote the timestamp \( s_i \).

We will prove that for any \( i \in \mathbb{N}, 0 \leq i \leq k \):

(a) For \( t = t_i \) and any integer \( t \in [t_i, t_{i+1}) \), \( B = \nu_i \) holds in state \( H[t] \).

(b) If \( i > 0 \), then the response of \( Op_i \) is the correct response for an operation execution of that type applied in state \( \nu_{i-1} \).

Part (b) implies the lemma, but we require both parts for induction. Now let \( S(i) \) denote parts (a)–(b) for a particular value of \( i \). Note that in \( \bar{H} \), the state of \( B \) is changed only by an execution of line 312, which is an atomic step that defines the timestamp of an operation execution (on the target object) in \( \bar{H} \). Therefore, the state of \( B \) does not change between atomic steps \( t_i \) and \( t_{i+1} \) in \( H \). This, in turn, implies that to prove part (a) of \( S(i) \), it suffices to prove that \( B = \nu_i \) holds in state \( H[t_i] \) — and that is all we do in the inductive step that follows.

For \( S(0) \), (a) follows from the initialization of \( B \) to \( \nu_0 \), and (b) holds trivially. Now for any \( i \), \( 0 < i \leq k \), suppose that \( S(i-1) \) holds, and consider \( S(i) \). We proceed by cases on how the timestamp \( s_i \) was obtained.

**Case A:** \( Op_i \) falls under Definition 10.11 (a). In this case, \( Op_i \) is a \( 0p-R() \) operation execution, which is purely read-like, and so \( \nu_i = \nu_{i-1} \).

It follows from our construction of \( \bar{H} \) (Definitions 10.11 and 10.14) that \( t_{i-1} < t_i \), and so \( B = \nu_{i-1} \) in state \( H[t_{i-1}] \) by \( S(i-1) \) (a). This implies that \( B = \nu_i \) in state \( H[t_i] \) because \( \nu_i = \nu_{i-1} \) and step \( t_i \) does not change the state of \( B \) (since it applies a purely read-like atomic step on \( B \)). Thus, \( S(i) \) (a) holds. \( S(i) \) (b) also holds because the response of \( Op_i \) is the response of a \( B.0p-R() \) in step \( t_i \), which is applied when \( B \) (an object of type \( \tau \)) is in state \( \nu_{i-1} \).

**Case B:** \( Op_i \) falls under Definition 10.11 (b). In this case, \( Op_i \) is a \( 0p-R() \) operation execution, which is purely read-like, and so \( \nu_i = \nu_{i-1} \).

If \( t_i = t_{i-1} \) then it follows from \( S(i-1) \) (a) that \( B = \nu_{i-1} \) in state \( H[t_i] \), which implies that that \( B = \nu_i \) in state \( H[t_i] \) since \( \nu_i = \nu_{i-1} \). Thus, \( S(i) \) (a) holds. For \( S(i) \) (b), note that since \( 0p-R() \) is purely read-like, the state of \( B \) is the same in \( H[t_i-1] \) as in \( H[t_i] \), in particular \( B = \nu_{i-1} = \nu_i \) in both. Thus, by line 310 the
response of \( Op_i \) is the response of a \( B.0p-R() \) in step \( t_i \), which is applied when \( B \) (an object of type \( \tau \)) is in state \( \nu_{i-1} \).

If \( t_i > t_{i-1} \), we proceed as in Case A.

**Case C:** \( Op_i \) falls under Definition 10.11 (d). In this case, \( Op_i \) is a \( 0p-W() \) operation execution, and \( p_i \) applies \( 0p-W() \) to \( B \) in step \( t_i \) of \( H \).

Since the timestamp of \( Op_i \) is of the form \((t_i,0)\), it follows that \( t_{i-1} < t_i \), and so \( B = \nu_{i-1} \) in state \( H[t_i-1] \) by \( S(i-1) \) (a). This implies that \( B = \nu_i \) in state \( H[t_i] \) because step \( t_i \) applies \( 0p-W() \) to \( B \) (an object of type \( \tau \)) in state \( \nu_{i-1} \). Thus, \( S(i) \) (a) holds. \( S(i) \) (b) also holds because the response of \( Op_i \) is the response of \( p_i \)'s \( B.0p-W() \) in step \( t_i \), which is applied when \( B \) is in state \( \nu_{i-1} \).

\[ \square \]

**Theorem 10.18.** The implementation \( I_U^4 \) of type \( \tau \) for \( N \) processes described in this section satisfies Specifications 6.1 (linearizability) and 6.2 (termination). Furthermore, each operation execution on the target object incurs \( O(1) \) RMRs in the DSM model under Condition 10.2. Finally, \( I_U^4 \) satisfies the locality property in the DSM model with respect to the designated process \( p_{\text{special}} \).

**Proof.**

**Specification 6.1 (linearizability):** This follows directly from Lemmas 10.15, 10.16 and 10.17.

**Specification 6.2 (termination):** Consider any fair history \( H \) of \( I_U^4 \). Calls to the pseudo-lock functions \( \text{Pseudo-Enter} \) and \( \text{Pseudo-Exit} \) terminate by Specification 4.3 since Condition 4.1 holds with respect to any pseudo-lock accessed in \( H \) by Lemma 10.9 (a), and since any process that acquires a pseudo-lock in any block \( x \) eventually calls \( x \triangleright \text{Pseudo-Exit}() \) the pseudo-lock functions terminate. Calls to \( \text{AllocBlock}() \) terminate by Specification 5.2. Since there are no busy-wait loops (outside of the pseudo-lock) and since all other statements only read or write registers, it follows that each execution of the access procedures defined for \( I_U^4 \) terminates.

**\( O(1) \) RMR complexity under Condition 10.2:** It suffices to consider operation executions by non-special processes because we show below that operation executions by \( p_{\text{special}} \) incur zero RMRs. Calls to \( \text{AllocBlock}() \) and operations on the block manager incur \( O(1) \) RMRs each by our assumption on the RMR complexity of these. Calls to
the pseudo-lock functions \texttt{Pseudo-Enter()} and \texttt{Pseudo-Exit()} also incur $O(1)$ RMRs by the RMR complexity of the pseudo-lock in the DSM model (Theorem 4.6) and by Lemma 10.9 (a). Accesses to the block field \textit{val} incur at most one RMR each, and there are at most two per operation execution. Finally, consider RMRs incurred while applying operations to the base object $B$, which is obtained using $I^3_U$. It follows from Lemma 10.10 and the algorithm that in histories of $I^3_U$, at most two non-special processes at a time can be applying an operation on $B$, namely one at line 312 and one at line 315. Consequently, in any such history Condition 10.7 holds with respect to $B$, and so each operation on $B$ incurs $O(1)$ RMRs by Theorem 10.18.

**Locality to $p_{\text{special}}$ in the DSM model:** This follows from the locality of the base object $B$ to $p_{\text{special}}$ (a property of its implementation $I^3_U$ – see Theorem 10.8) since the access procedures for $p_{\text{special}}$ access no other shared object.

\section*{10.5 Conclusion}

In this chapter we presented several universal constructions for shared objects that are locally accessible in the DSM model with respect to some designated process $p_{\text{special}}$. Each implementation has worst-case $O(1)$ RMR complexity per operation under certain conditions. Implementations $I^1_U$, $I^2_U$ and $I^3_U$ are the simplest conceptually, and have bounded space complexity as presented in this chapter. Implementation $I^4_U$ uses more complex synchronization techniques based on ideas from earlier chapters, and requires additional memory management (as described in Chapters 11–12) to bound space complexity.

Our implementations are applicable in the following scenarios: $I^1_U$ can be used if there are only two processes. $I^2_U$ can be used if there are $N$ processes but only $p_{\text{special}}$ and one non-special process at a time access the shared object (see Condition 10.5), otherwise it is not linearizable. $I^3_U$ can be used in all situations, but it guarantees $O(1)$ RMR complexity only if at most one non-special process at a time accesses the shared object (see Condition 10.7). $I^3_U$ has $O(\log N)$ RMR complexity per operation and remains local to $p_{\text{special}}$ otherwise. Finally, $I^4_U$ can be used in all situations, but it guarantees $O(1)$ RMR complexity only if at most $O(1)$ non-special process at a time apply operation executions of types that are not purely read-like (see Condition 10.2). $I^4_U$ has $O(\log N)$ RMR complexity per operation and remains local to $p_{\text{special}}$ otherwise.
Chapter 11

Bounding Space Complexity

In this chapter, we consider memory management in our ECAS implementations. (An analogous approach can be applied to the universal construction presented in Chapter 10.) We begin with a complete analysis that applies to the basic implementation presented in Chapter 6 and the writable ECAS implementation in Chapter 8. We then deal separately with the implementations that satisfy locality (see Chapter 7) in Chapter 12.

As discussed in Chapter 5, the state of the implemented ECAS object is recorded using units of memory called blocks that contain shared objects and instances of subroutines used for synchronization. Each time the state of the target object changes, the process whose ECAS operation execution caused the state change allocates a new block, and designates that block as current. Processes determine and change the current block by performing `getCurBlock` and `chgCurBlock` operations, respectively, on a block manager object. Processes allocate blocks using a block allocator, implemented by the function `AllocBlock()`. Any block returned by the allocator is fresh, meaning that every shared object in the block is in its initial state. For technical reasons, we relax the definition of a fresh block slightly in this chapter:

**Definition 11.1.** A block is fresh if every shared object in the block is in its initial state, except possibly `V` (which stores the value of the ECAS object, see Figure 6.5), `writer` (which records the ID of the process that created the block, see Figures 6.5 and 8.2) or `B` (which stores the underlying ECAS object used in the writable implementation, see Figure 8.2). If a block is not fresh then we call it dirty.

Since the fields `V`, `writer` and `B` are initialized explicitly after a block is allocated and before it becomes current (see Figures 6.6 and 8.2), it is sufficient for our purposes that
the block allocator return blocks that are fresh according to Definition 11.1 rather than the original definition from Chapter 5, where every field in a block must be in its initial state when the block is returned by the allocator.

We now introduce some new terminology related specifically to memory management.

**Definition 11.2.** For any history $H$ of our ECAS implementations, for any state $s$ that occurs during $H$, for any process $p$, and for any block $x$, we say that $p$ holds a strong (respectively weak) reference to $x$ in state $s$ if $p$ has made a call to `getCurBlock()` (respectively `AllocBlock()`) with response $x$, and has not yet finished executing the access procedure during which this call occurred. We say that $p$ holds a reference to $x$ in state $s$ if it holds a strong or weak reference to $x$ in state $s$.

The life-cycle of a block $x$ in our implementations of ECAS, as described up to this point, is illustrated in Figure 11.1 and consists of the following stages:

(a) Initially, $x$ resides in a pool of blocks maintained by the block allocator internally, and $x$ is fresh (see Definition 11.1). Whenever `AllocBlock()` is called, it removes a fresh block from the pool and returns the address of that block.

(b) After a process $p$ allocates $x$, $p$ tries to make $x$ current by calling `chngCurBlock(..., x)` on the block manager. If this call succeeds, $x$ becomes current. Otherwise, this call fails and $x$ is never accessed again, and remains fresh despite some of its fields having been overwritten by $p$ (see Definition 11.1).

(c) Once $x$ becomes current, it may be accessed repeatedly by any process and any of its fields may change from their initial states.

(d) If $x$ became current, it may eventually cease to be current. In that case, processes may continue to access block $x$ even after it is no longer current. For example, if process $p$ is executing an ECAS operation execution on the target object during which `getCurBlock` returns block $x$, and process $q$ concurrently executes an ECAS operation execution that changes the current block from $x$ to $y$, then $p$ may continue to access $x$ until it completes its ECAS operation execution on the target object.

(e) After $x$ ceases to be current, eventually no process holds a reference to $x$, and after that point no process accesses $x$ ever again. This is the final stage in the life-cycle of a block.
Figure 11.1: The lifecycle of a block.
The key to bounding the space complexity of our implementations of ECAS is to bound the total number of blocks needed. Since each block contains only a bounded number of shared objects, and our implementations use a constant number of other shared objects, this technique alone is sufficient for bounding space complexity. As regards the size of the registers underlying these shared objects, most require $O(\log N)$ bits, where $N$ is the maximum number of processes. The only exceptions are when a register stores an element from the domain $\mathcal{U}$ (of values of the ECAS object), and when a register stores a block address, but the latter quantity also becomes bounded once we bound the number of possible blocks. (As we show later in Theorem 11.33, $O(N)$ blocks suffice, and so block addresses can be represented using $O(\log N)$ bits.)

In order to bound the total number of blocks, we follow an approach similar in spirit to the popular technique of garbage collection using reference counting. In our context, we use the following definition of reference count:

**Definition 11.3.** For any history $H$ of our ECAS implementations, for any state $s$ that occurs during $H$, and for any block $x$, the reference count of $x$, denoted $\text{ref}(x)$, in state $s$ is the sum of:

- the number of processes that hold a strong reference to $x$ in state $s$ (see Definition 11.2); and
- the number of processes that hold a weak reference to $x$ in state $s$ (see Definition 11.2); and
- an indicator variable equal to one if $x$ is current in state $s$ and zero otherwise.

In our ECAS implementations, if block $x$ is returned by `AllocBlock()` and $\text{ref}(x)$ reaches zero subsequently, then block $x$ can be recycled – made fresh and returned once again by `AllocBlock()`. (We argue this more formally in Theorem 11.12 below.) The fact that we can recycle any block $x$ for which $\text{ref}(x) = 0$ motivates the following definition:

**Definition 11.4.** A block $x$ is recyclable if and only if $\text{ref}(x) = 0$ holds.

Recyclable blocks fall into two categories: those that became current after being allocated, and those that did not. To distinguish between the two types of recyclable blocks we introduce the following definition:

**Definition 11.5.** For any history $H$ of our ECAS implementations and any block $b$, a call to `AllocBlock()` with response $b$ is called visible if block $b$ becomes current after it
is returned and before \( \text{ref}(b) \) reaches zero subsequently. Otherwise it is called invisible.

Once the reference count for a block reaches zero, the manner in which the block is recycled depends on whether the call to \( \text{AllocBlock()} \) that (last) returned the block was visible or invisible. In the case of an invisible call, say by process \( p \), the block remains fresh and can be returned again safely the next time \( p \) calls \( \text{AllocBlock()} \). (This property is the motivation behind Definition 11.1.) On the other hand, in the case of a visible call the block may be dirty and must be reset before it can be reused. (We describe how to do this in Section 11.2.)

Because blocks allocated through visible calls to \( \text{AllocBlock()} \) must be reset before they can be reused, we need some way to identify such blocks once they become recyclable. We do this by using additional state information but without actually recording \( \text{ref}(x) \) as that would require an \( O(1) \)-RMR atomic counter, which in this context cannot be implemented using reads and writes. Instead, we deduce indirectly that \( \text{ref}(x) = 0 \) holds. This is still challenging in light of our \( O(1) \)-RMRs constraint, and we will present in Section 11.1 special algorithms for doing so. The high-level idea is that the ECAS implementation provides information to the block allocator by calling a special function \( \text{RecycleBlock()} \). The pair of functions \( \text{AllocBlock()} \) and \( \text{RecycleBlock()} \) then constitutes the allocator. (We describe how \( \text{RecycleBlock()} \) is called in Section 11.1.)

We now state a new specification for the block allocator that replaces Specifications 5.1–5.2 from Chapter 5. We make the specification more general than necessary for our purposes in this chapter in anticipation of Chapter 12, where multiple functions analogous to \( \text{RecycleBlock()} \) are used. We refer collectively to such functions as \textit{recycling functions}:

\textbf{Definition 11.6.} The recycling functions are: in Chapter 11 – \( \text{RecycleBlock()} \); in Section 12.3.2 – \( \text{RecycleBlock()} \), \( \text{RecycleBlockS()} \) and \( \text{RecycleBlockNS()} \).

\textbf{Specification 11.7.} For any history:

(a) \( \text{AllocBlock()} \) returns the address \( x \) of a fresh block such that \( \text{ref}(x) = 1 \) just after \( x \) is returned; and

(b) a call to \( \text{AllocBlock()} \) or one of the recycling functions (see Definition 11.6) accesses a block \( y \) only if \( \text{ref}(y) = 0 \).

\textbf{Specification 11.8.} For any fair history, each call to \( \text{AllocBlock()} \) or one of the recycling functions (see Definition 11.6) terminates.
Regarding any block $x$ returned by $\text{AllocBlock}()$, Specification 11.7 (a) states that $x$ is not current and no process other than the caller holds a reference to it. Part (b) ensures that when some block $y$ undergoes recycling, no process may observe $y$ in an inconsistent state as it is reset because at that point no process holds a reference to $y$.

We now state in Conditions 11.9–11.11 the assumptions underlying the design of the block allocator. Conditions 11.9 and 11.10 are etiquettes that processes must follow in accessing blocks, and in calling $\text{AllocBlock}()$ and $\text{RecycleBlock}()$, respectively. Condition 11.11 captures the relevant properties of reference counts.

**Condition 11.9.** For any history $H$, accesses to blocks satisfy the following:

(a) whenever a process $p$ accesses a block $x$ outside of calls to $\text{AllocBlock}()$ and the recycling functions (see Definition 11.6), $p$ holds a (strong or weak) reference to $x$; and

(b) if an invisible call (see Definition 11.5) to $\text{AllocBlock}()$ by some process $p$ returns block $x$ and then an access to $x$ occurs outside of calls to $\text{AllocBlock}()$ and the recycling functions (see Definition 11.6) before $\text{ref}(x)$ reaches zero, then this access does not make $x$ dirty (see Definition 11.1).

**Condition 11.10.** For any history $H$, calls to $\text{AllocBlock}()$ and $\text{RecycleBlock}()$ satisfy the following:

(a) calls to $\text{RecycleBlock}()$ are made in mutual exclusion; and

(b) a process calls $\text{RecycleBlock}()$ only after a visible call to $\text{AllocBlock}()$ and the corresponding successful $\text{chngCurBlock}(x, y)$, and does so exactly once before making another call to $\text{AllocBlock}()$; and

(c) the $i$'th call to $\text{RecycleBlock}()$ is by the process $p$ that applies the $i$'th successful $\text{chngCurBlock}(x, y)$, and occurs after this call to $\text{chngCurBlock}(x, y)$.

**Condition 11.11.** For any history $H$, reference counts satisfy the following:

(a) for any block $x$, if some step of $H$ increases $\text{ref}(x)$ from zero to a positive value then in this step a call to $\text{AllocBlock}()$ returns $x$; and

(b) if the $i$'th block that becomes current (initial block for $i = 0$) is $x$, then $\text{ref}(x)$ reaches zero after $x$ becomes current (i.e., after the $i$'th successful $\text{chngCurBlock}(\ldots, \ldots)$ or after the initial state if $i = 0$) and before the $(i+N+1)$'th call to $\text{RecycleBlock}()$ is made.
Having stated Specifications 11.7–11.8, we are now ready to assert the principal claim of this chapter:

**Theorem 11.12.** Our implementations of ECAS (see Chapter 6, 7 and 8) satisfy Specifications 6.1 and 6.2 (under Condition 6.4) and have \( O(1) \) RMR complexity in the CC and DSM models, provided that the block allocator satisfies Specifications 11.7–11.8 (instead of Specifications 5.1–5.2), and provided that each call to \( \text{AllocBlock()} \) incurs \( O(1) \) RMRs.

**Proof.** If the block allocator satisfies Specifications 11.7–11.8 instead of Specifications 5.1–5.2, we can establish the correctness properties of our ECAS implementations with some straightforward changes to these implementations and their proofs of correctness. These pertain to linearizability only; RMR complexity is dealt with exactly as before.

The key change to the implementations is the introduction of a *logical block address*, which is a tuple of the form \((a, i)\) where \(a\) is a block address, now referred to as the *physical block address*, and \(i\) is an integer. Logical addresses are generated by \( \text{AllocBlock()} \), which returns \((a, i)\) if \(i\) calls to \( \text{AllocBlock()} \) have previously returned logical addresses of the form \((a, j)\) for some \(j\) (e.g., \(i = 0\) the first time \( \text{AllocBlock()} \) is called). Similarly, whenever the initial block address \(b_0\) is used for initialization in the ECAS implementation, we instead use the tuple \((b_0, -1)\). Logical addresses are manipulated by the access procedures in the modified implementations just as physical addresses were originally, except that we redefine the operator \( \triangleright \) slightly: if \(x = (a, i)\) is a logical block address and \(f\) is a field in a block, then we re-define \(x \triangleright f\) as the instance of field \(f\) in block \(a\).

A subtle point that deserves more explanation is how \( \text{AllocBlock()} \) computes the value of \(i\) when it returns \((a, i)\). Although we consider implementations based on reads and writes only, we can think of \(i\) as the value obtained from an atomic counter. This is reasonable because such an atomic counter in this particular context is only a history variable. That is, the counter records information that is used only in our proofs, and is never used for control flow.

The above modifications to the ECAS implementations ensure that when \( \text{AllocBlock()} \) satisfies Specification 11.7 (a) with respect to physical addresses, it satisfies Specification 5.1 (a) with respect to logical addresses. This is because, by definition, each logical address returned by \( \text{AllocBlock()} \) is unique and different from the logical address of the initial block. However, this property alone does not allow us to establish the correctness of our ECAS implementations simply by using the original proofs with the term “block
address” replaced by the term “logical address”. Instead, the proofs must be amended further to accommodate the possibility that blocks may be accessed by the allocator internally (i.e., during calls to \texttt{AllocBlock()} or \texttt{RecycleBlock()}), and the possibility that two processes may access the same physical block outside of calls to \texttt{AllocBlock()} and \texttt{RecycleBlock()} via distinct logical addresses.

We amend our proofs of correctness by applying the following general rule: whenever we asserted an invariant regarding the state of a block with (logical) address \(x\) in some history \(H\), we now assert the same invariant but only for the subhistory \(H'\) of \(H\) from the point when \(x\) is returned by \texttt{AllocBlock()} (or from the initial state if \(x\) is the initial block) to the point when \(\text{ref}(x)\) reaches zero subsequently. (The subhistory \(H'\) is well-defined because \texttt{AllocBlock()} returns each logical address at most once, as explained earlier.) For example, instead of claiming that some field of block \(x\) is not overwritten after a certain point in \(H\), we claim that this field is not overwritten until \(\text{ref}(x)\) reaches zero. Similarly, instead of claiming that some field of \(x\) has not been accessed at all before a certain point in \(H\), we claim that it has not been accessed up to that point but only from the moment when \texttt{AllocBlock()} returned \(x\).

It remains to justify why such modifications suffice. First, note that any access to block \(x\) outside of calls to \texttt{AllocBlock()} and \texttt{RecycleBlock()} occurs when \(\text{ref}(x) > 0\) by Condition 11.9 (a), and so it corresponds to a step in the subhistory \(H'\) by our definition of \(H'\) and since \texttt{AllocBlock()} returns \(x\) at most once. Thus, any invariant pertaining to the state of block \(x\) is irrelevant (from the perspective of the ECAS implementation) except as it affects the values read by steps in \(H'\). Next, to accommodate the possibility that block \(x\) may be accessed by the allocator internally (i.e., during calls to \texttt{AllocBlock()} or \texttt{RecycleBlock()}), we must ensure that this does not happen during the subhistory \(H'\) defined above. But this follows immediately from Specification 11.7 (b) because \(\text{ref}(x) > 0\) holds throughout the subhistory \(H'\) of \(H\). Similarly, the possibility that two processes may access the same physical block via distinct logical addresses outside of calls to \texttt{AllocBlock()} and \texttt{RecycleBlock()} is dealt with because such a pair of accesses cannot occur during \(H'\). To see this, suppose for contradiction that \(H'\) contains a step where the physical block \(b\) corresponding to the logical address \(x\) is accessed via another logical address \(y \neq x\). Note that by Condition 11.9 (a) some process holds a reference to \(y\) during such an access, and consider which of \(x\) and \(y\) was allocated first. If \(y\) was allocated before \(x\) then this must have occurred during \(H'\), which contradicts Specification 11.7 (a) because just after the call to \texttt{AllocBlock()} that returns \(y\) the
physical block $b$ has reference count greater than one. If $x$ was allocated before $y$ then $y$ must have been allocated during $H'$, and we arrive at the same contradiction with respect to the call to \texttt{AllocBlock()} that returns $x$. 

In the remainder of this chapter, we describe a block allocator that satisfies Specifications 11.7–11.8. We organize the presentation of this allocator as follows. First, in Section 11.1 we describe how the allocator is integrated with the ECAS implementation that uses it (i.e., how the subroutine \texttt{RecycleBlock()} is called) and why this modified implementation satisfies Conditions 11.9–11.11 (on which the allocator relies). Then, in Section 11.2 we describe efficient techniques for resetting blocks. Finally, in Section 11.3 we give implementations of \texttt{AllocBlock()} and \texttt{RecycleBlock()} that rely on Conditions 11.9–11.11 and apply the techniques from Section 11.2. To simplify presentation, we defer discussion of locality properties to Chapter 12.

\section{11.1 Modifications to ECAS Implementations}

In this section we describe how to integrate a block allocator that satisfies Specification 11.7 under Conditions 11.9–11.11 with our ECAS implementations (see Chapter 6, 7 and 8). To that end, we must describe how \texttt{RecycleBlock()} is called, and argue why Conditions 11.9–11.11 hold. In both cases we need to modify the ECAS implementations, and we do so by calling special subroutines at particular places in the access procedures. The new subroutines are called \texttt{BeforeOp()}, \texttt{MiddleOp}(x, y) and \texttt{AfterOp()}. Functions \texttt{BeforeOp()} and \texttt{AfterOp()} are called immediately before and after an operation execution on the implemented ECAS object, respectively. The information recorded by \texttt{BeforeOp()} and \texttt{AfterOp()} makes it possible to detect when a process holds a reference to some block (as opposed to being “in-between” operation executions on the target object). To ensure this we adopt the following convention: in the context of Definition 11.2, we consider that the call to \texttt{AfterOp()} is not part of the access procedure, and so a process executing \texttt{AfterOp()} does not hold a reference to any block. (How we classify \texttt{BeforeOp()} is not important, but for concreteness we consider that a call to this function is part of the access procedure.) Function \texttt{MiddleOp}(x, y) is called immediately after a successful \texttt{chgCurBlock}(x, y) and serves several purposes, discussed shortly, related to Conditions 11.10 (a) and 11.11 (b). Figure 11.2 shows as an example how the subroutines \texttt{BeforeOp()}, \texttt{MiddleOp}(x, y) and \texttt{AfterOp()} are called in the ECAS access
procedure from Chapter 6 (Figure 6.6). The algorithms for the subroutines are then shown in Figures 11.3 and 11.4.

The subroutines use internally several shared variables. A process $p$ uses the shared Boolean array $Active[1..N]$ to indicate whether it is executing an access procedure (line 324 of $BeforeOp()$ and line 325 of $AfterOp()$). In addition, in $AfterOp()$ process $p$ sometimes synchronizes with another process $w$ that is waiting for $p$ inside $MiddleOp(x,y)$. This synchronization is effected using two shared arrays: $WaiterID[1..N]$ and $WaitFlag[1..N][1..N]$: process $p$ discovers $w$’s ID by reading $WaiterID[p]$ (line 326), and then signals $w$ by writing $WaitFlag[w][p]$ (line 328).

Function $MiddleOp(x,y)$ serves two purposes: (1) it effects a “waiting mechanism” at lines 336–346 which ensures that some other process releases any reference it may have to any block that became current before $y$ (including block $x$); and (2) it calls $RecycleBlock()$ at line 347. The waiting mechanism is needed to satisfy Condition 11.11 (b), and calls to $RecycleBlock()$ (made in a particular way) are needed to satisfy Condition 11.10 (a).

The fact that $MiddleOp(x,y)$ may be executed concurrently by multiple processes leads to several complications. First, consider the waiting mechanism mentioned above. In this mechanism, some process $p$ that calls $MiddleOp(x,y)$ (i.e., the waiter) selects one other process $q$ and, upon detecting that $q$ has a pending operation execution on the target object, $p$ waits for $q$ to call $AfterOp()$ and by doing so release any references $q$ holds to blocks. The obvious problem that arises when we allow multiple “sessions” of the waiting mechanism to occur in parallel is the possibility of deadlock: if $p$ is waiting inside $MiddleOp(x,y)$ for $q$ to call $AfterOp()$, then $q$ may instead enter a busy-wait loop (e.g., by calling $MiddleOp(y,\ldots)$ and waiting therein for $p$). Another more subtle problem is that several waiters may wait for the same process $q$, which complicates synchronization. Finally, when processes execute $MiddleOp$ in parallel, calls to $RecycleBlock()$ at line 347 must be serialized as needed for Condition 11.10 (a).

To prevent deadlock inside the waiting mechanism, a process executes statements identical to lines 326–329 of $AfterOp()$ at the beginning of each call to $MiddleOp(x,y)$ (lines 331–334). As a side-effect, this may break the waiting mechanism because when $p$ finishes waiting for $q$, $q$ may still hold a reference to some block. However, our solution to the other problems identified above ensures that $q$ may only hold a reference that it acquired after $p$ made block $y$ current prior to calling $MiddleOp(x,y)$, which is sufficient for our purposes. The solution to the other problems is to serialize executions of the
Function ECAS(isSC, cmp, new)

BeforeOp()

\[ d := M.\text{getCurBlock}() \]

\[ ... \]

\[ \text{if } (\text{isSC} = \text{true} \land \text{read}(d \triangleright \text{Linked}[\text{PID}]) = \text{false}) \lor (\text{isSC} = \text{false} \land \text{cmp} \neq \text{old}) \text{ then} \]

\[ ... \]

\[ \text{else if isSC} = \text{false} \land \text{cmp} = \text{new} \text{ then} \]

\[ ... \]

\[ \text{else} \]

\[ // \text{Try to execute successful operation execution that changes the state.} \]

\[ d' := \text{AllocBlock}() \]

\[ ... \]

\[ \text{winner} := M.\text{chngCurBlock}(d, d') \]

\[ \text{if winner} = \text{PID} \text{ then} \]

\[ \quad \text{MiddleOp}(d, d') \]

\[ ... \]

\[ \text{return } (...) \]

AfterOp() // Executed immediately after return, and not part of access procedure (by convention).

Figure 11.2: Instrumented ECAS access procedure corresponding to Figure 6.6.
waiting mechanism as well as calls to `RecycleBlock()`, and moreover to order these executions in a particular way. (Note that although this removes one source of deadlock, it creates another source and so lines 331–334 of `MiddleOp(x, y)` remain necessary.)

The part of `MiddleOp(x, y)` that is serialized comprises lines 336–348, and is referred to as the “critical section” in Figure 11.4 and our subsequent analysis. Since using a mutual exclusion algorithm in this context would be too costly in terms of RMRs, we rely on a more specialized mechanism that takes advantage of the manner in which `MiddleOp(x, y)` is called. The key idea is that if `y` is the `i`'th block that becomes current, then the `i`'th execution of the critical section occurs during a call to `MiddleOp(x, y)` for some block `x` (see Lemma 11.15). Since `x` is the `(i − 1)`'st block that became current, to ensure mutual exclusion it suffices to wait for the execution of the critical section during the call to `MiddleOp(..., x)` to end before the critical section during the call to `MiddleOp(x, y)` begins.

A process gains access to the critical section by calling function `MiddleOp-EnterCS(x)` at line 335, and releases the critical section by calling `MiddleOp-ExitCS(y)` at line 349. To implement these subroutines, we introduce three shared variables in each block: `CSCleared`, `CSWaiter`, and `CSWaitFlag[1..N]`. Inside `MiddleOp-EnterCS(x)`, process `p` writes its ID to `x ⊲ CSWaiter` (line 350), and then ensures that another process `q` has called `MiddleOp-ExitCS(x)` by testing `x ⊲ CSCleared` (line 351) and possibly waiting for `x ⊲ CSWaitFlag[p] = true` (line 352). Then, inside `MiddleOp-ExitCS(y)` process `p` assigns `y ⊲ CSCleared = true` (line 354) to indicate that it has cleared the critical section, `p` then tries to discover the ID of a process `q` that is waiting for it (in a call to `MiddleOp-EnterCS(y)` by reading `y ⊲ CSWaiter` (line 355); if it discovers such a process, `p` signals `q` by assigning `y ⊲ CSWaitFlag[q] = true` (line 357).

The waiting mechanism mentioned earlier is inside the critical section, at lines 336–346 of `MiddleOp(x, y)`, and works as follows. The global variable `ctr` is used as a modular counter to determine the ID of the next process to be wait for (lines 336–337). Now let `c` denote the process ID selected here, and let `p` denote the caller of `MiddleOp(x, y)`. If `c ≠ p` (line 338) then `p` proceeds as follows. First, `p` resets the Boolean spin variable `WaitFlag[p][c]` (line 339) in case `p` needs to wait for `c` later on. Next, `p` announces its ID to `c` by writing `WaiterID[c]` (line 340). Then, `p` tries to determine whether `c` has a pending operation execution on the target object by testing `Active[c] = true` (line 341). If this holds, `p` must ensure that `c` either calls `AfterOp()` or is waiting to enter the critical section inside a call to `MiddleOp-EnterCS(...)` at line 335 of `MiddleOp`. (In either case,
c no longer holds a reference to any block it may have acquired before \(p\) made \(y\) current, as mentioned earlier and shown later in Lemma 11.16.) To that end, \(p\) waits for \(c\). To a first approximation, the condition for which \(p\) should wait is \(\text{WaitFlag}[p][c] = \text{true}\) (line 343), because \(c\) will cause this to hold when it executes \(\text{AfterOp}()\) or lines 331–334 of \(\text{MiddleOp}\). However, because \(c\) may have already executed lines 331–334 of \(\text{MiddleOp}\) before \(p\) wrote its ID to \(\text{WaiterID}[c]\) at line 340, \(p\) needs another way to detect that \(c\) is contending for the critical section. To that end, we introduce a shared Boolean array \(\text{CSContender}[1..N]\). Process \(c\) sets \(\text{CSContender}[c]\) upon beginning a call to \(\text{MiddleOp}\) (line 330) and then resets it just before exiting the critical section (line 348). To synchronize with \(c\), \(p\) tests \(\text{CSContender}[c]\) at line 342, and waits on \(\text{WaitFlag}[p][c]\) at line 343 only if it read \text{true}.

In the remainder of this section we analyze the properties of instrumented implementations of ECAS, which are our original ECAS implementations (see Chapter 6, 7 and 8) modified by introducing calls to \(\text{BeforeOp}(), \text{MiddleOp}(x,y)\) and \(\text{AfterOp}()\) as described earlier (see Figure 11.2). We show that the instrumented implementations satisfy the same correctness properties as the original implementations, and in addition satisfy Conditions 11.9–11.11. In our analysis, we assume that the block allocator satisfies Specifications 11.7–11.8.

**Lemma 11.13.** For any history of the instrumented ECAS implementations, a successful call to \(\text{chngCurBlock}(x, y)\) changes the current block from \(x\) to \(y\). Furthermore, \(y \neq x\).

**Proof.** We showed in Lemma 6.6 (and its analogs in later chapters, namely Lemmas 7.6, 8.2 and 8.8) that a successful call to \(\text{chngCurBlock}(x, y)\) in histories of the original ECAS implementations changes the current block from \(x\) to \(y\). The same property can be shown for the instrumented implementations as the same approach as in the proof of Theorem 11.12. To see why \(y \neq x\), consider an operation execution by some process \(p\) where a successful \(\text{chngCurBlock}(x, y)\) occurs. Note that \(x\) is a block \(p\) obtained by calling \(\text{getCurBlock}()\) on the block manager, and \(y\) is a block that \(p\) later obtained by calling \(\text{AllocBlock}()\). Thus, just after \(\text{AllocBlock}()\) returns \(y\) to \(p\), \(p\) holds a strong reference to \(x\) and a weak reference to \(y\). If \(x = y\) then this implies \(\text{ref}(y) \geq 2\), which contradicts Specification 11.7 (a).

**Lemma 11.14.** For any history of the instrumented ECAS implementations, for any distinct processes \(p\) and \(q\), and for any block \(x\), process \(q\) does not make \(x\) current or
Declarations for BeforeOp(), MiddleOp(x, y), and AfterOp().

Shared variables: (global)
- Active[1..N], WaiterID[1..N], CSContender[1..N] – Boolean arrays, all elements initially false
- WaitFlag[1..N][1..N] – array of Boolean, all elements initially false, elements [i][1..N] local to process i in the DSM model
- ctr – integer, initially 1

Shared variables: (per-block)
- CSWaiter – process ID or ⊥, initially ⊥
- CSCleared – Boolean, initially true for the initial block, and false for all others
- CSWaitFlag[1..N] – array of Boolean, initially all false, element i local to process i in the DSM model

Private variables: (per-process)
- c, q, wi – process ID or ⊥, uninitialized

Function BeforeOp()
// Called before every operation execution on the target object.
324 write Active[PID] := true

Function AfterOp()
// Called after every operation execution on the target object.
325 write Active[PID] := false
326 wi := read(WaiterID[PID])
327 if wi ≠ ⊥ then
328 | write WaitFlag[wi][PID] := true
329 end

Figure 11.3: Subroutines BeforeOp() and AfterOp().
Function MiddleOp(x, y)
Input: x – address of block returned by last call to getCurBlock()
Input: y – address of block returned by last call to AllocBlock()

// Enter critical section.
write CSContender[PID] := true
wi := read(WaiterID[PID])
if wi ≠ ⊥ then
  write WaitFlag[wi][PID] := true
end
MiddleOp-EnterCS(x)

// Critical section begins.
c := read(ctr)
write ctr := (c mod N) + 1
if c ≠ PID then
  // Ensure process c does not hold a strong reference to block x
  // or another block that became current earlier.
  write WaitFlag[PID][c] := false
  write WaiterID[c] := PID
  if read(Active[c]) = true then
    if read(CSContender[c]) = false then
      await WaitFlag[PID][c] = true
    end
  end
end
RecycleBlock()
write CSContender[PID] := false
// Critical section ends.
MiddleOp-ExitCS(y)

Function MiddleOp-EnterCS(x)
write x ⊲ CSWaiter := PID
if read(x ⊲ CSCleared) = false then
  await
  x ⊲ CSWaitFlag[PID] = true
end

Function MiddleOp-ExitCS(y)
write y ⊲ CSCleared := true
q := read(y ⊲ CSWaiter)
if q ≠ ⊥ then
  write y ⊲ CSWaitFlag[q] := true
end

Figure 11.4: Subroutine MiddleOp(x, y).
execute a step inside MiddleOp(..., x) while p holds a reference to x.

Proof. Suppose, for contradiction, that q does make x current or takes a step inside MiddleOp(..., x) while p holds a reference to x in some history H of the instrumented ECAS implementations. Without loss of generality, suppose that |H| is minimal and consider the subhistory H′ of H starting with the step where AllocBlock() returns x to q and ending just before q makes x current or calls MiddleOp(..., x). (Recall that process q must allocate x before making it current or calling MiddleOp(..., x) by the structure of the access procedures.) By definition of H′ and by Specification 11.7 (a), ref(x) = 1 holds just after the first step of H′. Thus, x is not current and q is the only process that holds a reference to it at that point. Furthermore, ref(x) ≥ 1 holds until the end of H′ by the minimality of |H|. Consequently, no process other than q acquires a weak reference to x in H′ by Specification 11.7 (a), and no process acquires a strong reference to x in H′. Since p does not hold a reference to x at the beginning of H′ (because only q does, as explained earlier) this contradicts p holding such a reference at the end of H′. □

Lemma 11.15. For any history of the instrumented ECAS implementations, let bᵢ denote the i’th block that becomes current (initial block for i = 0), let pᵢ denote the process that makes it current (i.e., executes the i’th successful chngCurBlock on the block manager) and let Cᵢ denote the call to MiddleOp by pᵢ following this chngCurBlock(..., bᵢ) (if such a call exists at all). Then:

(a) lines 336–348 of MiddleOp are executed in mutual exclusion and the i’th execution of these lines occurs during Cᵢ; and

(b) lines 336–348 during Cᵢ are executed with c mod N = i mod N.

Proof.
Part (a): Let (xᵢ, yᵢ) denote the arguments of Cᵢ. First, note that since Cᵢ has the same arguments as the preceding chngCurBlock by pᵢ, and so yᵢ = bᵢ. Furthermore, xᵢ = bᵢ₋₁ by Lemma 11.13. Now to prove part (a), it suffices to show that for any i ≥ 1, if pᵢ is at line 336 with xᵢ = bᵢ₋₁ during Cᵢ, then pᵢ₋₁ already completed line 348 with yᵢ₋₁ = bᵢ₋₁ during Cᵢ₋₁. Suppose for contradiction that this does not hold for some i. Consider pᵢ’s execution of MiddleOp-EnterCS(bᵢ₋₁) at line 335 just before it reaches line 336 during Cᵢ. Since pᵢ completed line 335, it follows that pᵢ found
\( b_{i-1} \trianglerighteq \text{CSCleared} = \text{true} \) at line 351 or \( b_{i-1} \trianglerighteq \text{CSWaitFlag}[\text{PID}] = \text{true} \) at line 352. Since both variables are initialized to \text{false}, it follows that some process wrote \text{true} to one of these variables. By the algorithm, this occurred either at line 354 or at line 357 during a call to \text{MiddleOp-ExitCS}(b_{i-1}). (It did not occur during a call to \text{AllocBlock()} or \text{RecycleBlock()} by Specification 11.7 (b) and the fact that \text{ref}(b_{i-1}) > 0 \) from the moment \( b_{i-1} \) was allocated until \( p_i \) reads \( b_{i-1} \trianglerighteq \text{CSCleared} = \text{true} \) during \( C'_i \); first \( p_{i-1} \) holds a weak reference to \( b_{i-1} \), then \( p_{i-1} \) makes \( b_{i-1} \) current, then \( p_i \) obtains a strong reference to \( b_{i-1} \) and holds that reference until it completes \( C_i \).) In particular, this happened in the subhistory \( H' \) starting just after \( p_{i-1} \) allocates \( b_{i-1} \) and ending just before \( p_i \) reads \( b_{i-1} \trianglerighteq \text{CSCleared} \) during \( C_i \). If \( p_{i-1} \) makes the call \( C'_{i-1} \) in \( H' \), then \( p_{i-1} \) completes line 348 of \( C_{i-1} \) before \( p_i \) reads \( b_{i-1} \trianglerighteq \text{CSCleared} \) during \( C_i \), which contradicts our earlier hypothesis. Otherwise, \( p_{i-1} \) does not take a step inside \text{MiddleOp(...,} b_{i-1}) \text{in} \ H', and neither does any other process by Lemma 11.14 because in this case \( p_{i-1} \) holds a weak reference to \( b_{i-1} \) throughout \( H' \). But this contradicts the earlier observation that some process writes \( b_{i-1} \trianglerighteq \text{CSCleared} \) in \( H' \) during a call to \text{MiddleOp-ExitCS}(b_{i-1}), which is only called from \text{MiddleOp(...,} b_{i-1}).

**Part (b):** Note that \( c \) is the value read at line 336 of \text{MiddleOp} from \text{ctr}, which is incremented (modulo \( N \)) at line 337 during every execution of lines 336–348. Thus, it follows from part (a) and the initialization of \text{ctr} to 1 that \( c \mod N = i \mod N \) holds when lines 336–348 are executed for the \( i \)’th time (i.e., during \( C_i \)).

**Lemma 11.16.** For any history of the instrumented ECAS implementations, suppose that process \( p \) executes lines 338–346 of \text{MiddleOp}(x,y) with \( c = q \) for some process \( q \neq p \). If \( q \) acquired a strong reference to a block before \( p \)’s successful \text{chngCurBlock}(x,y) preceding \( p \)’s call to \text{MiddleOp}(x,y), then \( q \) releases this strong reference before \( p \) reaches line 346.

**Proof.** Suppose, for contradiction, that \( q \) continuously holds its strong reference to \( b \) until \( p \) reaches line 346. When \( q \) obtains its reference to \( b \) (before \( p \)’s \text{chngCurBlock}(x,y)), \( q \) has already completed a call to \text{BeforeOp()} but not yet made the subsequent call to \text{AfterOp()}. Furthermore, by our assumption about \( q \), it does not make such a call to \text{AfterOp()} until after \( p \) reaches line 346. Thus, it follows from the algorithm for \text{BeforeOp()} and \text{AfterOp()} (see lines 324 and 325) that \( \text{Active}[q] = \text{true} \) holds during \( p \)’s entire execution of lines 338–346 under consideration. In particular, \( p \) reads \( \text{Active}[q] = \text{true} \) at line 341 and then completes lines 342–344 with \( c = q \).
We will show that

\[ \text{CSContender}[q] = \text{false} \] holds during \( p \)'s execution of lines 338–346. \hspace{1cm} (11.1)

Suppose otherwise. Then \( q \) calls \texttt{MiddleOp} during the operation execution where it acquires its reference to block \( b \) under consideration. Now define \( b_i, p_i \) and \( C_i \) as in Lemma 11.15, let \( C_j \) denote \( q \)'s call to \texttt{MiddleOp}, and let \( C_{j'} \) denote \( p \)'s call to \texttt{MiddleOp} where it observes \( \text{CSContender}[q] = \text{true} \) at line 342 while \( q \) is executing \( C_j \). By Lemma 11.15 (a) either \( q \) completes line 349 of \( C_j \) before \( p \) starts line 336 of \( C_{j'} \) (in which case \( j < j' \)) or \( p \) completes line 349 of \( C_{j'} \) before \( q \) starts line 336 of \( C_j \) (in which case \( j' < j \)). In the former case, because \( q \) executes line 348 before it reaches line 349, we have that \( \text{CSContender}[q] = \text{false} \) while \( p \) is at lines 338–346, contrary to our supposition. Therefore \( j' < j \).

Since \( j' < j \), the following sequence of events occurs: \( q \) obtains its strong reference to block \( b \), then \( p \) applies a successful \texttt{chngCurBlock}(\( b_{j'-1}, b_{j'} \)), and then \( q \) applies a successful \texttt{chngCurBlock}(\( b_{j-1}, b_j \)), where \( b_{j-1} = b \) by the structure of the access procedures. By Lemma 11.13, this means that after \( q \) obtains its reference to block \( b \), \( b \) ceases to be current (by the action of \( p \)'s \texttt{chngCurBlock} or an earlier one), and then becomes current again just before \( q \)'s \texttt{chngCurBlock}. In particular, some process other than \( q \) makes \( b \) current while \( q \) holds a reference to \( b \), which contradicts Lemma 11.14.

Thus, \( \text{Active}[q] = \text{true} \) and \( \text{CSContender}[q] = \text{false} \) during \( p \)'s execution of lines 338–346, and in particular lines 342–344. Since \( p \) first assigns \( \text{WaitFlag}[p][q] \) to \text{false} at line 339, and then completes line 343 (since we assume it reaches line 346), it follows that some process assigns \( \text{WaitFlag}[p][q] = \text{true} \) during \( p \)'s execution of lines 342–344. By the algorithm, it must be \( q \) that does so, either at line 328 of \texttt{AfterOp()} or at line 333 of \texttt{MiddleOp}. The first alternative is impossible because \( q \) does not call \texttt{AfterOp()} until after \( p \) completes line 346. The second alternative is also impossible, because in that case \( \text{CSContender}[q] = \text{true} \) when \( q \) is at line 333 of \texttt{MiddleOp} while \( p \) is at lines 342–344, contradicting statement (11.1) above.

**Theorem 11.17.** For any history of the instrumented ECAS implementations, Condition 11.9 holds.

**Proof.**

**Part (a):** This follows from the structure of the access procedures in the instrumented
ECAS implementations, whereby a process accesses a block \( x \) during an operation execution only if it has acquired a reference to \( x \) by calling \texttt{getCurBlock()} or \texttt{AllocBlock()}.

**Part (b):** If an invisible call to \texttt{AllocBlock()} returns block \( x \), then the process \( p \) that makes this call holds a weak reference to \( x \) until it completes its operation execution, and does not make \( x \) current during that operation execution. Furthermore, \( p \) only modifies fields \( V, B \) or \textit{writer} of \( x \) while \( p \) holds its weak reference to \( x \). Consequently, no other process can obtain a strong reference to \( x \), or a weak reference by Specification 11.7 (a), and so by part (a) no other process accesses \( x \). Thus, it follows from Definition 11.1 that \( x \) is fresh and \( \text{ref}(x) = 0 \) when \( p \) releases its weak reference to \( x \) under consideration. \( \square \)

**Theorem 11.18.** For any history of the instrumented ECAS implementations, Condition 11.10 holds.

*Proof.*

**Part (a):** This follows from Lemma 11.15 (a) and the fact that \texttt{RecycleBlock()} is only called at line 347 of \texttt{MiddleOp}.

**Parts (b):** This follows from the structure of the access procedures in our instrumented ECAS implementations and from Definition 11.5.

**Part (c):** This follows from Lemma 11.15 (a), the algorithm for \texttt{MiddleOp} (line 347), and our assumption that a process calls \texttt{MiddleOp}(\( x, y \)) after a successful \texttt{chngCurBlock}(\( x, y \)). \( \square \)

**Theorem 11.19.** For any history of the instrumented ECAS implementations, Condition 11.11 holds.

*Proof.*

**Part (a):** The reference count for a block can increase only if a process \( p \) obtains a reference to \( x \) or makes \( x \) current (see Definition 11.3). Now suppose that \( \text{ref}(x) = 0 \) beforehand. If \( p \) obtains a strong reference to \( x \), then \( x \) is current beforehand, which contradicts \( \text{ref}(x) = 0 \) holding then. If \( p \) makes \( x \) current then \( p \) holds a weak reference to \( x \) beforehand, which again contradicts \( \text{ref}(x) = 0 \) beforehand. Thus, if \( \text{ref}(x) \) increases from zero then some process obtains a weak reference to \( x \) (through a call to \texttt{AllocBlock()} with response \( x \)).

**Part (b):** Let \( H \) be the history under consideration and suppose that \( x \) is the \( i \)'th block that becomes current in \( H \). Suppose also that the \((1+i+N)\)'th call to \texttt{RecycleBlock()} is made in \( H \). (The order of calls to \texttt{RecycleBlock()} is well-defined by Lemma 11.15 (a)
and the fact that RecycleBlock() is only called at line 347 of MiddleOp.) Let $H'$ be the subhistory of $H$ beginning just after the $i$’th successful chngCurBlock (i.e., the one that made $x$ current), and ending just before the $(1 + i + N)$’th call to RecycleBlock() is made. We must show that $\text{ref}(x)$ reaches zero in $H'$. Suppose for contradiction that $\text{ref}(x) > 0$ holds continuously throughout $H'$.

It follows from Lemma 11.15 (a) and the definition of $H'$ that $H'$ contains $N + 1$ consecutive executions of lines 336–348 of MiddleOp, from the $i$’th one to the $(i + N)$’th one (inclusive). Let $E_i, ..., E_{i+N}$ denote these. We will now show that $x$ is not current continuously throughout $H'$. Since there are $N + 1$ calls to MiddleOp in $H'$, there is one process that executes two of these, and so by the structure of the access procedures this process applies a successful chngCurBlock between these two calls in $H'$. (Although there is one successful chngCurBlock in $H$ for every call to MiddleOp, up to $N$ of these may occur before $H'$, and so there may only be one in $H'$.) Since $H'$ contains at least one successful chngCurBlock, by Lemma 11.13 block $x$ ceases to be current after the first of these, as wanted.

To complete the proof, it suffices to show that after $x$ ceases to be current $\text{ref}(x)$ eventually decreases to zero in $H'$. Suppose for contradiction that this does not happen. Then some process $p$ continuously holds a strong reference to $x$ in $H'$ from some state before $x$ ceases to be current, until the end of $H'$. This follows from three observations:

First, once $x$ ceases to be current, $x$ cannot become current again by Lemma 11.14 because we assume $\text{ref}(x) > 0$ throughout $H'$. Second, while $x$ is not current, no process can acquire a strong reference to it by Definition 11.2. Third, no process can acquire a weak reference to $x$ while $\text{ref}(x) > 0$ by Specification 11.7 (a). Thus, $p$’s reference to $x$ is strong and $p$ obtains it before the $(i + 1)$’st successful chngCurBlock in $H$, hence before $E_{i+1}$ by Lemma 11.15 (a).

So far we have shown that $H'$ contains $E_i, ..., E_{i+N}$, and that $p$ continuously holds a strong reference to $x$ from the beginning of $E_{i+1}$ until the end of $E_{i+N}$ in $H'$. We will now show that $p$ does not execute $E_j$ for $i + 1 \leq j \leq i + N$. If $p$ did, then $p$ also executes the $j$’th successful chngCurBlock in $H$ by Lemma 11.15 (a), at which time $p$ holds a strong reference to the $j$’th block that becomes current and to no other block, by the structure of the access procedures of the instrumented ECAS implementations. In particular, $p$ does so after $E_i$ hence while holding its strong reference to $x$, which implies that $j = i$ because $x$ is the $i$’th block that becomes current. But this contradicts $i + 1 \leq j \leq i + N$. Thus, $E_{i+1}, ..., E_{i+N}$ are executed by processes other than $p$, and
by Lemma 11.15 (b) one such execution occurs with \( c = p \), say \( E_{j'} \) by process \( z \neq p \). Before \( z \) reaches line 346 during \( E_{j'} \), \( p \) releases its reference to \( x \) under consideration by Lemma 11.16, which contradicts \( p \) holds its strong reference to \( x \) from the beginning of \( E_{i+1} \) until the end of \( E_{i+N} \) in \( H' \).

**Theorem 11.20.** For any fair history \( H \) of the instrumented ECAS implementations, each execution of \( \text{BeforeOp()} \), \( \text{MiddleOp} \), and \( \text{AfterOp()} \) eventually terminates.

**Proof.** Since \( \text{BeforeOp()} \) and \( \text{AfterOp()} \) involve no unbounded loops, each call to these functions terminates. To show that \( \text{MiddleOp} \) terminates, it suffices to show that the busy-wait loops at line 343 (of the \( \text{MiddleOp} \) mainline) and line 352 (of the subroutine \( \text{MiddleOp-EnterCS} \)) eventually terminate. Suppose otherwise.

**Case A:** Some process \( q \) loops forever at line 343, repeatedly reading \( \text{WaitFlag}[q][z] = \text{false} \) for some process ID \( z \). Note that when \( q \) last executed line 341, it read \( \text{Active}[z] = \text{true} \), and so process \( z \) at the time was between line 324 of \( \text{BeforeOp()} \) and line 325 of \( \text{AfterOp()} \). Since the ECAS implementation satisfies Specification 6.2 (termination) before being instrumented with the subroutines \( \text{BeforeOp()} \), \( \text{AfterOp()} \) and \( \text{MiddleOp} \) (see Theorem 11.12), it follows that \( z \) eventually completes either lines 326–329 of \( \text{AfterOp()} \), or lines 331–334 of \( \text{MiddleOp} \).

**Subcase A-i:** \( z \) completes lines 326–329 of \( \text{AfterOp()} \). Since \( z \) was between line 324 of \( \text{BeforeOp()} \) and line 325 of \( \text{AfterOp()} \) when \( q \) last executed line 341 of \( \text{MiddleOp} \), \( z \) reads \( \text{WaiterID}[z] \) at line 326 of \( \text{AfterOp()} \) after \( q \) writes its ID to \( \text{WaiterID}[z] \) at line 340 of \( \text{MiddleOp} \). Furthermore, no process overwrites \( \text{WaiterID}[z] \) after \( q \) writes it and before \( z \) reads it because this can only happen at line 340 of \( \text{MiddleOp} \), and yet no process can execute this line by Lemma 11.15 (a) and our assumption in Case A that \( q \) does not progress past line 343 after writing \( \text{WaiterID}[z] \). Thus, \( z \) reads \( q \)'s ID from \( \text{WaiterID}[z] \) at line 326 of \( \text{AfterOp()} \), and subsequently writes \( \text{true} \) to \( \text{WaitFlag}[q][z] \) at line 328 of \( \text{AfterOp()} \). Since this variable can be assigned \( \text{false} \) only by \( q \) and only at line 339, this contradicts the hypothesis of Case A.

**Subcase A-ii:** \( z \) completes lines 331–334 of \( \text{MiddleOp} \). Since \( z \) was between line 324 of \( \text{BeforeOp()} \) and line 325 of \( \text{AfterOp()} \) when \( q \) last executed line 341 of \( \text{MiddleOp} \), \( z \) subsequently assigns \( \text{CSContender}[z] = \text{true} \) at line 330. Furthermore, the value of this variable does not change after that point because only \( z \) can change it and only at line 348 of \( \text{MiddleOp} \), which \( z \) cannot reach by Lemma 11.15 (a) since in Case A \( q \) loops forever at line 343 after completing line 341. Consequently, \( z \)'s assignment of
\(CSContender[z] = \text{true}\) at line 330 follows \(q\)'s read of \(CSContender[z] = \text{false}\) at line 342. Then \(z\)'s read of \(WaiterID[z]\) at line 331 follows \(q\)'s write of \(WaiterID[z]\) at line 340. Arguing as in Subcase A-i, it follows that \(z\)'s read returns \(q\)'s ID, \(z\) then writes \text{true} to \(WaitFlag[q][z]\) at line 333 of \text{MiddleOp}, and no process assigns \text{false} to \(WaitFlag[q][z]\) subsequently, which contradicts the hypothesis of Case A.

**Case B:** some process \(q\) loops forever at line 352 of \text{MiddleOp-EnterCS}(x) during a call to \text{MiddleOp}(x, y) for some blocks \(x\) and \(y\), repeatedly reading \(x \triangleright CSWaitFlag[q] = \text{false}\). Suppose that \(q\)'s successful \text{chngCurBlock}(x, y) prior to its call to \text{MiddleOp} under consideration is the \(i\)'th one in \(H\). Then letting \(b_j\) denote the \(j\)'th block that becomes current, as in Lemma 11.15, it follows by Lemma 11.13 that \(x = b_{i-1}\) and \(y = b_i\). Without loss of generality suppose that \(q\) is chosen so that \(i\) is minimal. If \(i = 1\) then \(x\) is the initial block, where \(x \triangleright \text{CSCleared}\) is initialized to \text{true}. Furthermore, no process assigns \(x \triangleright \text{CSCleared} = \text{false}\) before \(q\) reads it because this never happens outside of calls to \text{AllocBlock()} and \text{RecycleBlock()} by the algorithm, and similarly it never happens during such calls by Specification 11.7 (b) and since \(\text{ref}(x) > 0\) from initialization until \(q\) reads \(x \triangleright \text{CSCleared}\) (i.e., first \(x\) is current, then \(q\) holds a strong reference to \(x\)). Thus, \(q\) reads \text{true} from \(x \triangleright \text{CSCleared}\) at line 351 during its call to \text{MiddleOp-EnterCS()}, which contradicts \(q\) branching to line 352 as we assume in Case B.

Thus, \(i > 1\) and some process \(p\) made block \(x = b_{i-1}\) current by executing a successful \text{chngCurBlock}(b_{i-2}, b_{i-1}). Since \(H\) is fair, \(p\)'s next step after this \text{chngCurBlock} is to call \text{MiddleOp}(b_{i-2}, b_{i-1}). During this call, \(p\) completes a call to \text{MiddleOp-EnterCS}(b_{i-2}) at line 335 by minimality of \(i\), and eventually calls \text{MiddleOp-ExitCS}(b_{i-1}) at line 349. (It does not loop forever at line 343 by Case A.) During this call to \text{MiddleOp-ExitCS}(b_{i-1}) \(p\) assigns \(b_{i-1} \triangleright \text{CSCleared} = \text{true}\) at line 354, and as in the case when \(b_{i-1}\) is the initial block we can show that no process subsequently assigns \(b_{i-1} \triangleright \text{CSCleared} = \text{false}\) until \(\text{ref}(b_{i-1})\) reaches zero, which does not occur until after \(q\) reads this variable at line 351 during its call to \text{MiddleOp-EnterCS}(b_{i-1}). Since in Case B \(q\) branches to line 352, this implies that \(q\) reads \(b_{i-1} \triangleright \text{CSCleared}\) at line 351 before \(p\) writes it at line 354. Consequently, \(q\) writes its ID to \(b_{i-1} \triangleright \text{CSWaiter}\) at line 350 before \(p\) reads it at line 355. Between \(q\)'s write and \(p\)'s read, no process overwrites this variable outside of calls to \text{AllocBlock()} and \text{RecycleBlock()} because this can only occur at line 350 during a call to \text{MiddleOp}(..., b_{i-1}), which \(q\) has already completed and which no other process can complete by Lemma 11.14 and because \(q\) is in \text{MiddleOp}. Similarly no process overwrites this variable inside \text{AllocBlock()} or \text{RecycleBlock()} because \(\text{ref}(b_{i-1}) > 0\) while \(q\) is
in MiddleOp and by Specification 11.7 (b). Thus, \( p \) reads \( q \)'s ID from \( b_{i-1} \triangleright CSWaiter \) at line 355, and then assigns \( b_{i-1} \triangleright CSWaitFlag[q] = \text{true} \) at line 357. After this write and until \( \text{ref}(b_{i-1}) \) reaches zero, no process overwrites this variable outside of calls to AllocBlock() and RecycleBlock() by the algorithm, and no process does so inside AllocBlock() or RecycleBlock() either by Specification 11.7 (b). Since \( \text{ref}(b_{i-1}) > 0 \) continues to hold while \( q \) loops forever at line 352, the fact that \( b_{i-1} \triangleright CSWaitFlag[q] = \text{true} \) holds contradicts the hypothesis of Case B and \( x = b_{i-1} \).

**Theorem 11.21.** For any history of the instrumented ECAS implementations, each execution of BeforeOp(), MiddleOp, and AfterOp() incurs \( O(1) \) RMRs in the CC and DSM models provided that each call to RecycleBlock() incurs \( O(1) \) RMRs.

**Proof.** In the DSM model, the lemma follows from the structure of the functions under consideration, our assumption on the RMR complexity of RecycleBlock(), and the locality of \( \text{WaitFlag}[p][1..N] \) and \( \text{CSWaitFlag}[p] \) (in any block) to process \( p \).

In the CC model, the lemma follows similarly provided that each execution of the busy-wait loops at line 343 (of MiddleOp) and line 352 (of MiddleOp-EnterCS) incurs \( O(1) \) RMRs. First, consider the loop line 343, which terminates when a process \( p \) reads \( \text{WaitFlag}[p][c] = \text{true} \). Here \( p \) incurs at most one RMR at the first access to \( \text{WaitFlag}[p][c] \), and at most one more after each subsequent write (by another process) to \( \text{WaitFlag}[p][c] \), until \( p \) reads \( \text{true} \). By the algorithm, only \( p \) can write \( \text{false} \) to \( \text{WaitFlag}[p][c] \), namely at line 339 of MiddleOp. Consequently, \( p \) incurs at most one more RMR reading \( \text{WaitFlag}[p][c] \) at line 343, for a total of two.

Next, consider the loop at line 352. For any block \( x \) and process \( p \), note that by the algorithm \( x \triangleright \text{CSWaitFlag}[p] \) can only be written at line 357 of MiddleOp-ExitCS outside of calls to AllocBlock() and RecycleBlock(), and the value written there is \( \text{true} \). Furthermore, since \( \text{ref}(x) > 0 \) while a process is at line 352, \( x \triangleright \text{CSWaitFlag}[p] \) is not overwritten inside AllocBlock() or RecycleBlock() by Specification 11.7 (b). Thus, any execution of line 352 incurs at most two RMRs as in the case of line 343. □

**Theorem 11.22.** The instrumented implementations of ECAS satisfy Specifications 6.1 and 6.2 (under Condition 6.4) and \( O(1) \) RMR complexity in the CC and DSM models provided that each call to AllocBlock() and RecycleBlock() incurs \( O(1) \) RMRs.

**Proof.** The instrumented implementations differ from the original ones only in that calls to BeforeOp(), AfterOp() and MiddleOp are made in specific places, and that the block
allocator satisfies Specifications 11.7–11.8 instead of Specifications 5.1–5.2. Since the new subroutines do not access any of the shared variables accessed by the original access procedures, Specification 6.1 follows from Theorem 11.12. Similarly, Specification 6.2 and $O(1)$ RMR complexity follow from Theorems 11.12, 11.20 and 11.21.

11.2 Techniques for Resetting Blocks

In this section, we describe how blocks can be reset efficiently once they have been identified as recyclable (see Definition 11.4). The main challenge here, in light of our $O(1)$ RMR constraint, is due to each block containing a super-constant number of shared registers. In the implementations of ECAS under consideration, this number is bounded by $cN^k$ for integers $c$ and $k$. In particular, $c \leq 10$ and $k = 2$ hold if we use the name consensus algorithm from Chapter 3 and the leader election algorithm from [11] as building blocks underlying the pseudo-lock. Since $k > 0$, this precludes the very simple solution of resetting a block by resetting each register in the block to its initial value as this would incur too many RMRs for our purposes. (In our ECAS implementations, in the worst case one block is allocated per operation execution, and so the allocator must be able to produce $k$ blocks using only $O(k)$ RMRs.)

Blocks can be reset more efficiently by employing more sophisticated algorithms. One approach is to identify which registers in a block differ from their initial value, and reset only those. Since each process accesses $O(1)$ registers in each block, this bounds the RMR complexity of resetting a block at $O(N)$ irrespective of $k$. The disadvantage of this approach is that additional synchronization is needed. For example, if a block is reset entirely by one process, then that process must know which others accessed the block, and which registers they accessed. On the other hand, if each process is responsible for resetting the registers that it accessed, then synchronization is needed to determine when each process has completed its share of the work.

To keep the memory management algorithms simple and efficient, we adopt yet another approach, which minimizes additional synchronization. In this approach, we define two kinds of resets: soft and hard. In a soft reset, processes use a novel technique that resets a block using only $O(1)$ RMRs. This technique is efficient but can only be performed a bounded number of times. To deal with the latter problem, we also employ a hard reset, which uses the naive technique described earlier (i.e., resets each register individually). By combining soft and hard resets, it is possible to reduce the amortised
RMR complexity for resetting blocks to $O(1)$ RMRs per block. Finally, taking advantage of the fact that the work of resetting blocks can be done in mutual exclusion (recall Condition 11.10 (a)), it is straightforward to complete this work using an incremental approach that incurs only $O(1)$ RMRs worst-case per operation execution on the target object. (This requires that hard resets be spread out over multiple operation executions.)

Supporting soft resets requires that we change fundamentally the way processes access blocks. This is because to achieve $O(1)$ RMR complexity an average, a soft reset cannot overwrite each register in a block, or even $O(N)$ registers. Instead, a soft reset informs processes indirectly that the block has been reset. To that end, we tag blocks and individual registers in a block with version numbers. For each block, we introduce a special block version register, initially zero and denoted $\mathit{x\triangleright blockVersion}$ for block $x$. For every other shared register in a block, we embed a special field in that register that stores a register version, also initially zero. When a process accesses a block for the first time during some operation execution on the target object, it first reads the block version, say $\mathit{verBlock}$. A particular register in the block (other than the block version register) is then accessed as follows during the same operation execution on the target object: To write a value $v$ to a register, a process instead writes a tuple $(\mathit{verBlock}, v)$ to the same register, indicating that the register was last overwritten when the block had version $\mathit{verBlock}$. To read a register, a process reads from the register a tuple $(\mathit{verBlock}', v)$, and then returns $v$ if $\mathit{verBlock} = \mathit{verBlock}'$, or else returns the initial value of the register. It follows easily that when shared registers in a block are accessed this way, the pertinent correctness properties of our ECAS implementations are preserved. This includes RMR complexity and locality properties, provided that in the DSM model we make the block version register local to the designated process $p_{special}$.

Soft and hard resets can be combined into an efficient technique for resetting blocks. Recall that each block contains at most $cN^k$ shared registers (other than the block version). The first $cN^k$ times a block is reset, we use only soft resets, which causes the block version to grow to a $\lceil k \log N \rceil$-bit value. Then, we fall back on a hard reset, which sets the block version to zero and reverts every other register to $(0, v_0)$ where $v_0$ is the initial value for that register. After that, we start once again applying soft resets, and repeat the cycle as needed. Since a block is reused $\Theta(N^k)$ times in each a cycle, and $\Theta(N^k)$ RMRs are performed to apply all the soft and hard resets in one cycle, it follows that the amortized cost of resetting a block is $O(1)$ RMRs.

The writable ECAS implementations presented in Chapter 8 require special consid-
eration because they use blocks that contain shared objects other than registers. Recall that in those implementations, each block contains a field \( B \) that is itself an ECAS object. Although we assume that \( B \) is implemented using read/write registers only (at least to prove Theorem 9.1), maintaining register versions for such registers is conceptually complex because \( B \) performs its own memory management and may maintain its own register versions. For simplicity, we instead treat \( B \) as a “black box” object and we do not assign version numbers to \( B \) or the underlying registers. Instead, we rely on a new operation type called \( \text{Reset}() \), which sets \( V \) and \( \text{Linked}[1..N] \) to their initial values (as specified for type \( \tau_{\text{ECAS}} \)). A \( \text{Reset}() \) is applied on \( B \) during both soft and hard resets on a block in the writable ECAS implementation. (Statements that do so appear in Figure 11.7 at line 364 and line 375.) Every \( \text{Reset}() \) operation is classified as write-like.

Although \( \text{Reset}() \) is similar in spirit to the “initialization” operation type described in Chapter 8, there is one important difference: \( \text{Reset} \) may be applied on the target object in any state, whereas in Chapter 8 we assumed that \( \text{Linked}[1..N] \) has at most one entry with the value \text{true} (i.e., the entry whose index is the caller’s ID) just before an initialization operation is applied. Consequently, we need a new mechanism to implement \( \text{Reset} \). Since Condition 11.10 (a) ensures that \( B \) is accessed in mutual exclusion during a \( \text{Reset} \) operation execution, the \( \text{Reset} \) operation type can be implemented using a very simple access procedure: the caller first calls \( \text{getCurBlock} \) to determine the current block, say \( x \), then allocates a block \( y \), initializes \( y \uparrow V \) to the initial value for type \( \tau_{\text{ECAS}} \), and finally calls \( \text{chngCurBlock}(x, y) \) to make \( y \) current. (The subroutines \( \text{BeforeOp} \), \( \text{MiddleOp} \) and \( \text{AfterOp} \) defined in Section 11.1 are not called during this access procedure because there is no need for this.) It follows easily that this access procedure resets \( B \) correctly and incurs \( O(1) \) RMRs in the worst case, as wanted.

### 11.3 Block Allocator

In this section, we describe how the functions \( \text{AllocBlock}() \) and \( \text{RecycleBlock}() \) of the block allocator are implemented. We model executions of these functions using a concurrent system \( S \), and we show that histories of \( S \) satisfy Specifications 11.7 and 11.8 provided that Conditions 11.9–11.11 hold. We also show that each call to \( \text{AllocBlock}() \) or \( \text{RecycleBlock}() \) incurs \( O(1) \) RMRs. Because the specifications are phrased in terms of reference counts (see Definition 11.3), it is implied that histories of \( S \) are generated in the context of some implementation \( I \) that uses blocks with a particular structure.
However, to keep our analysis general we simply assume that each process may call AllocBlock() and RecycleBlock() repeatedly and also access blocks outside of such calls in a manner that satisfies Conditions 11.9–11.11 (see Theorems 11.17–11.19).

Pseudo-code for the functions AllocBlock() and RecycleBlock() is presented in Figures 11.6–11.7. We adopt the following notational convention in Figure 11.7 to describe how hard resets are carried out: we assume that the registers in each block are numbered from 1 to some upper limit denoted BlockSize, and we denote by $x \triangleright register(i)$ the $i$'th register in block $x$. (In the context of writable ECAS implementations, we exclude registers underlying the object $B$ because such registers receive special treatment as explained at the end of Section 11.2.)

The block allocator internally maintains several global shared variables. These include several disjoint collections of blocks: FreshBlocks is a double-ended queue of fresh blocks (see Definition 11.1), DirtyBlocks is a FIFO queue of blocks that must be hard-reset before they can be reused, RecentBlocks is a FIFO queue containing the last $N + 1$ blocks that became current (padded with fresh blocks initially), and NextBlock[1..N] is an array of pointers to fresh blocks where NextBlock[p] is the block that $p$ has either last allocated or will allocate next. Linearizable implementations of the queues FreshBlocks, DirtyBlocks and RecentBlocks can be implemented easily using read/write registers at a cost of $O(1)$ RMRs per operation because calls to RecycleBlock() are made in mutual exclusion by Condition 11.10 (a). (E.g., we can use doubly-linked lists with pointers to the head and tail and a running total of the number of elements.)

Calls to RecycleBlock() move blocks among the allocator’s data structures according to the pattern shown in Figure 11.5, while maintaining their number and distinctness (see Lemma 11.23). In each call to RecycleBlock(), one or more of the following occurs:

- one block is moved from NextBlock[1..N] to RecentBlocks at line 360
- one block is moved from FreshBlocks to NextBlock[1..N] at line 361
- one block is moved from RecentBlocks to either FreshBlocks at lines 362 and 366 or to DirtyBlocks at lines 362 and 368
- one block is moved from DirtyBlocks to FreshBlocks at lines 377–378

Function AllocBlock() simply reads and returns the value of NextBlock[PID] (line 359), and relies on a subsequent execution of RecycleBlock() to “reload” the value of NextBlock[PID]
Declarations

Shared variables: (global)
- FreshBlocks – double-ended queue of block addresses, initially contains $N + 4$ fresh blocks different from the initial block
- DirtyBlocks – FIFO queue of block addresses, initially empty
- NextBlock[1..N] – array of block addresses, initially holds the addresses of $N$ distinct fresh blocks different from those in FreshBlocks and from the initial block
- numRegsReset – integer, initially zero
- RecentBlocks – FIFO queue of block addresses, initially contains the addresses of $N$ distinct fresh blocks different from those in FreshBlocks and NextBlock[1..N] and from the initial block, followed by the initial block (i.e, $N + 1$ blocks in total)

Shared variables: (per-block)
- blockVersion – block version number, integer from 0 toBlockSize – 1, initially 0

Private variables: (per-process)
- $b, b'$ – block address, uninitialized
- $i$ – integer, uninitialized

Figure 11.5: Possible transitions of blocks among data structures of the allocator.

Figure 11.6: Block allocator – part 1 (declarations).
Figure 11.7: Block allocator – part 2 (subroutines `AllocBlock()` and `RecycleBlock()`).
with the address of a fresh block from \( \text{FreshBlocks} \) (line 361). This is not needed following a invisible call to \( \text{AllocBlock()} \) because in that case the block returned remains fresh after the caller releases its weak reference to that block (see Definition 11.1 and Condition 11.9 (b)).

Function \( \text{RecycleBlock()} \) begins by recording a history of up to \( N + 2 \) blocks that became current recently using the queue \( \text{RecentBlocks} \) (line 360). Next, it “reloads” the allocator for the calling process by extracting a fresh block from \( \text{FreshBlocks} \) and writing its address to \( \text{NextBlock}[\text{PID}] \) (line 361). Then, it extracts a recyclable block \( b \) from \( \text{RecentBlocks} \) at line 362. (The fact that this block is recyclable is related to Condition 11.11 (b), as we explain later in the proof of Lemma 11.29.) If it is possible to soft-reset block \( b \), then this is done at lines 363–365, and in that case \( b \) is enqueued at the head of \( \text{FreshBlocks} \) at line 366. We could also enqueue \( b \) at the tail, but enqueuing at the head simplifies our analysis later in the proof of Lemma 11.26.) Otherwise, \( b \) has already been soft-reset the maximum the number of times, and so it requires a hard reset. In that case, \( b \) is enqueued at the tail of \( \text{DirtyBlocks} \) at line 368. Next, \( \text{RecycleBlock()} \) tries to perform a hard reset on some block. The block chosen is the head of \( \text{DirtyBlocks} \), which is determined at line 371. If such a block \( b' \) exists (see line 370) then \( \text{RecycleBlock()} \) performs a partial hard reset on \( b' \) at lines 372–373, where by “partial” we mean that only one register in \( b' \) is reset. The value of the variable \( \text{numRegsReset} \) is read at line 372 to determine the index of the next register to be reset. The test at line 374 then determines whether this completes the hard reset, and if so then block \( b' \) is moved to \( \text{FreshBlocks} \) and \( \text{numRegsReset} \) is reset at lines 377–379. Otherwise, the shared variable \( \text{numRegsReset} \) is incremented at line 381 so that the next execution of \( \text{RecycleBlock()} \) can continue hard-resetting block \( b' \). (The head element of \( \text{DirtyBlocks} \) does not change until it is dequeued because elements are enqueued into \( \text{DirtyBlocks} \) only at the tail, namely at line 368.)

The correctness properties of \( \text{AllocBlock()} \) and \( \text{RecycleBlock()} \) are stated in the lemmas below.

**Lemma 11.23.** Let \( L \) denote the set of \((3N + 5)\) blocks in the data structures of the allocator at initialization. For any history \( H \) of \( S \) where Condition 11.10 holds, if \( \text{FreshBlocks} \) is non-empty at each execution of line 361 of \( \text{RecycleBlock()} \) then the elements of the data structures \( \text{FreshBlocks}, \text{DirtyBlocks}, \text{RecentBlocks} \) and \( \text{NextBlock}[1..N] \) are elements of \( L \) in any state during \( H \). Furthermore, these elements are distinct in any state during
Chapter 11. Bounding Space Complexity

\( H \) where:

(a) no process is executing `RecycleBlock();` or
(b) some process is about to dequeue an element from `RecentBlocks` at line 362 of `RecycleBlock();` or
(c) some process is at line 370 of `RecycleBlock().`

**Proof.** Since the correctness properties under consideration are safety properties it suffices to consider finite \( H \). We proceed by induction on the length of the history \( H \).

**Basis:** \( |H| = 0 \). At initialization, `FreshBlocks` contains \( N + 4 \) blocks, `RecentBlocks` contains \( N + 1 \) blocks, `DirtyBlocks` is empty, and `NextBlock[1..N]` contains \( N \) blocks. The set \( L \) is the union of these blocks. Since the initial state satisfies clause (a) above, we must also show that these elements are distinct (i.e., that \( |L| = 3N + 5 \)). But this follows from initialization, and so the lemma holds.

**Induction step:** For any \( i \in \mathbb{N} \), suppose that the lemma holds when \( |H| = i \), and consider \( |H| = i + 1 \). We proceed by cases on the line of pseudo-code that generated step \( i + 1 \), noting that the lemma for \( H \) follows directly from the induction hypothesis (IH) unless step \( i + 1 \) changes the state of one of the data structures under consideration (i.e., `FreshBlocks`, `DirtyBlocks`, `RecentBlocks` and `NextBlock[1..N]`) or is a step that yields a state that falls under clauses (a)–(c). Let \( p \) be the process that applies step \( i + 1 \) and consider this step. We must show that if \( p \) enqueues or writes an element into one of the data structures, then this element is the address of a block from \( L \). Furthermore, in some cases we must show that no block is duplicated or lost between states classified under clauses (a)–(c) above.

**Case A:** \( p \) enqueues an element \( b \) into `RecentBlocks` at line 360. Since \( p \) read \( b \) from `NextBlock[p]` at line 360, it follows from the IH that \( b \in L \), and so \( b \) is a block address. Since \( p \) is at line 360 of `RecycleBlock()` in state \( H[i+1] \), it follows by Condition 11.10 (a) that state \( H[i + 1] \) does not fall under clauses (a)–(c), and so our analysis is complete.

**Case B:** \( p \) dequeues an element \( b \) from `FreshBlocks` (which we assume is not empty) at line 361. Since \( p \)'s step does not add or replace elements in any of the data structures under consideration, there is nothing to show regarding the block \( b \). Furthermore, since \( p \) is at line 361 in state \( H[i + 1] \), it follows as in Case A that state \( H[i + 1] \) does not fall under clauses (a)–(c), and so our analysis is complete.

**Case C:** \( p \) writes an element into `NextBlock[p]` at line 361. Since \( p \) obtained \( b \) from `FreshBlocks` at line 361, it follows from the IH that \( b \in L \), and so \( b \) is a block address.
Since step $i + 1$ yields a state that falls under clause (b) above, we must also consider distinctness. By Condition 11.10 (a), the last state before $H[i + 1]$ that falls under clauses (a)–(c) is one where $p$ is about to make its call to \texttt{RecycleBlock()}, say in step $j$. By the IH, the distinctness property holds for $H[j]$. Between $H[j]$ and $H[i + 1]$, $p$ is the only process that changes the state of the allocator’s data structures by Condition 11.10 (a), and $p$ simply permutes the elements as follows: $p$ moves one element from $\texttt{NextBlock}[p]$ to $\texttt{RecentBlocks}$ at line 360 and another from $\texttt{FreshBlocks}$ to $\texttt{NextBlock}[p]$ at line 361. Consequently, distinctness holds in $H[i + 1]$ as well.

**Case D:** $p$ dequeues an element $b$ from $\texttt{RecentBlocks}$ at line 362. First, note that $\texttt{RecentBlocks}$ is not empty just before step $i + 1$ because $p$ enqueued an element into $\texttt{RecentBlocks}$ earlier at line 360 and because between that step and step $i + 1$ no other process accessed by $\texttt{RecentBlocks}$ by the algorithm and Condition 11.10 (a). The analysis is completed as in Case B.

**Case E:** $p$ enqueues an element $b$ into $\texttt{FreshBlocks}$ at line 366. Since $p$ dequeued $b$ from $\texttt{RecentBlocks}$ line 362, it follows from the IH that $b \in L$, and so $b$ is a block address. Since step $i + 1$ yields a state that falls under clause (c) above, we must also consider distinctness. The analysis is analogous to Case C: distinctness is preserved because at lines 362–366 process $p$ merely moves one element from $\texttt{RecentBlocks}$ to $\texttt{FreshBlocks}$.

**Case F:** $p$ enqueues an element $b$ into $\texttt{DirtyBlocks}$ at line 368. The analysis is analogous to Case E.

**Case G:** $p$ dequeues an element $b$ from $\texttt{DirtyBlocks}$ at line 377. It follows from Condition 11.10 (a) and the success of the test at line 370 that $\texttt{DirtyBlocks}$ is not empty when $p$ is at line 377. The analysis is the same as in Case B.

**Case H:** $p$ enqueues an element $b$ into $\texttt{FreshBlocks}$ at line 378. Since $p$ obtained $b$ from $\texttt{DirtyBlocks}$ at line 377, it follows from the IH that $b \in L$, and so $b$ is a block address. Furthermore, since $p$ is at line 379 in state $H[i + 1]$, it follows as in Case A that state $H[i + 1]$ does not fall under clauses (a)–(c), and so our analysis is complete.

**Case I:** $p$ completes a call to \texttt{RecycleBlock()} (by executing an unsuccessful test at line 370, or by executing line 379 or line 381). Since step $i + 1$ yields a state that falls under clause (a) above, we must consider distinctness. The analysis is analogous to Case C: distinctness is preserved because at lines 370–383 process $p$ either did not change the state of the allocator’s data structures, or it merely moved one element from $\texttt{DirtyBlocks}$ to $\texttt{FreshBlocks}$ at lines 377–378. \(\square\)
Lemma 11.24. For any history of $S$ where Condition 11.10 holds, if $\text{FreshBlocks}$ is non-empty at each execution of line 361 of $\text{RecycleBlock()}$ then the queue $\text{RecentBlocks}$ contains exactly $N + 1$ elements at the beginning of each call to $\text{RecycleBlock()}$.

Proof. During each (complete) call to $\text{RecycleBlock()}$, one element is enqueued into $\text{RecentBlocks}$ at line 360 and one element is dequeued at line 362. This and Condition 11.10 (a) imply the lemma because $\text{RecentBlocks}$ contains $N+1$ elements initially.  

Lemma 11.25. For any history of $S$ where Condition 11.10 holds, if $\text{FreshBlocks}$ is non-empty at each execution of line 361 of $\text{RecycleBlock()}$ then whenever an element is enqueued into $\text{FreshBlocks}$ at line 378, this element is the address of a block $b$ where $b \triangleright blockVersion = 0$.

Proof. Lemma 11.23 applies since Condition 11.10 holds, and so we know that any element $b$ enqueued into $\text{FreshBlocks}$ is a block address. Next, note that the element enqueued into $\text{FreshBlocks}$ at line 378 was dequeued from the head of $\text{DirtyBlocks}$ at line 377, and so by Condition 11.10 (a) and the algorithm for $\text{RecycleBlock()}$ (lines 371 and 376) process $p$ wrote $b \triangleright blockVersion = 0$ at line 376. Furthermore, $b \triangleright blockVersion = 0$ holds until $p$ enqueues $b$ into $\text{FreshBlocks}$ at line 378 by Condition 11.10 (a) and the fact that the block version register (i.e., $b \triangleright blockVersion$) is never written outside of $\text{RecycleBlock()}$.  

Lemma 11.26. For any history of $S$ where Condition 11.10 holds, and for any call to $\text{RecycleBlock()}$, $\text{FreshBlocks}$ is never empty when a process is about to dequeue an element from $\text{FreshBlocks}$ at line 361.

Proof. Suppose for contradiction that the lemma is false and consider the shortest history $H$ at the end of which some process is at line 361 of $\text{RecycleBlock()}$ and $\text{FreshBlocks}$ is empty. Recall that calls to $\text{RecycleBlock()}$ are made in mutual exclusion by Condition 11.10 (a) and let $s_i$ denote the state just before the $i$’th such call. Since $\text{FreshBlocks}$ contains $N + 4$ elements in $s_1$ by initialization, and since each call to $\text{RecycleBlock()}$ changes the length of $\text{FreshBlocks}$ by at most one (see lines 361, 366 and 378), it follows that $\text{FreshBlocks}$ contains exactly $N + 3$ elements in some state $s_k$, $k > 0$. Without loss of generality, choose $k$ to be the maximum possible so that $\text{FreshBlocks}$ has $N + 3$ or fewer elements in every state $s_i$ for $i \geq k$. Let $s_{k'}$ denote the state just before the last call to $\text{RecycleBlock()}$ in $H$, where we assume $\text{FreshBlocks}$ is empty.
Now consider what happens in $H$ after state $s_k$. First, note that the hypotheses of Lemma 11.23 hold throughout $H$, and so the following properties hold in all the states in $s_i$ for $i \geq k$: First, all the elements of $\text{FreshBlocks}$, $\text{DirtyBlocks}$, $\text{RecentBlocks}$ and $\text{NextBlock}[1..N]$ are distinct block addresses from the set $L$ defined in the statement of Lemma 11.23. Second, the total number of elements in these data structures is constant. Third, since $\text{FreshBlocks}$ contains at most $N + 3$ blocks, which is at least one less than initially, the second point implies that $\text{DirtyBlocks}$, $\text{RecentBlocks}$ and $\text{NextBlock}[1..N]$ together contains at least one more block than initially, namely at least $2N + 2$ blocks. Since the number of elements in $\text{NextBlock}[1..N]$ is fixed at $N$ and since $\text{RecentBlocks}$ contains exactly $N + 1$ in every state $s_i$ by Lemma 11.24, this means $\text{DirtyBlocks}$ contains at least one block in every $s_i$ for $i \geq k$.

To derive a contradiction, we will derive two bounds on the total number of calls to $\text{RecycleBlock()}$ between states $s_k$ and $s_{k'}$ (i.e., $k' - k + 1$). Let $T$ denote this quantity, and let $X$ be the number of such calls where a block is enqueued into $\text{FreshBlocks}$ at line 378. First, consider how many blocks (counting with multiplicity) are moved from $\text{DirtyBlocks}$ to $\text{FreshBlocks}$ between states $s_k$ and $s_{k'}$. Since $\text{DirtyBlocks}$ is non-empty in that subhistory, as explained earlier, it follows from the algorithm for $\text{RecycleBlock()}$ (lines 370–383) and Condition 11.10 (a) that a block is enqueued into $\text{FreshBlocks}$ at line 378 at least once every $\text{BlockSize}$ calls to $\text{RecycleBlock()}$, and so $X \geq \lceil T/\text{BlockSize} \rceil \geq T/\text{BlockSize} - 1$, which implies:

$$T \leq (X + 1) \times \text{BlockSize} \quad (11.2)$$

Next, consider how many blocks (counting with multiplicity) are moved from $\text{RecentBlocks}$ to $\text{FreshBlocks}$ at line 366 between states $s_k$ and $s_{k'}$. Each time a block $b$ is enqueued into $\text{FreshBlocks}$ at line 366, the block version of $b$ is less than $\text{BlockSize} - 1$ at line 363, and is then incremented during the soft-reset at line 365. Now consider how many such soft resets must occur before $\text{FreshBlocks}$ becomes empty, which is at most $T$. First, note that any block $x$ that is in $\text{FreshBlocks}$, except possibly the head element, has $x \triangleright \text{blockVersion} = 0$ in any state $s_i$ for $i \geq k$. This is because any block enqueued at the tail of $\text{FreshBlocks}$ is enqueued at line 378 and has $x \triangleright \text{blockVersion} = 0$ by Lemma 11.25, and any block enqueued at the head of $\text{FreshBlocks}$ is enqueued at line 366 following the removal of the head element earlier at line 361. ($\text{FreshBlocks}$ does not change states between line 361 and line 366 by Condition 11.10 (a).) Thus, there are at least $N + 2$ (out of $N + 3$) blocks in $\text{FreshBlocks}$ in state $s_k$ with $\text{blockVersion} = 0$, and there are $X$
more such blocks enqueued later at line 378. Now it follows by a straightforward but somewhat tedious induction (similar to the proof of Lemma 11.23) that when a block \( x \) with \( \text{blockVersion} = 0 \) is in \( \text{FreshBlocks} \), it remains in \( \text{FreshBlocks} \), \( \text{NextBlock}[1..N] \) or \( \text{RecentBlocks} \) between \( s_k \) and \( s_{k'} \) until \( x \triangleright \text{blockVersion} \) reaches \( \text{BlockSize} - 1 \) (line 363), at which time \( x \) is enqueued into \( \text{DirtyBlocks} \) at line 368. (See Figure 11.5.) Thus, each of the \( N + 2 + X \) blocks with \( \text{blockVersion} = 0 \) that are either in \( \text{FreshBlocks} \) in \( s_k \) or are enqueued at the tail of \( \text{FreshBlocks} \) later (line 378) must be either soft-reset \( \text{BlockSize} \) times and enqueued into \( \text{DirtyBlocks} \) (line 368) between \( s_k \) and \( s_{k'} \), or else moved to \( \text{NextBlock}[1..N] \) after \( s_k \) and remain there until \( s_{k'} \), otherwise \( \text{FreshBlocks} \) is not empty in \( s_{k'} \). Since there are at most \( N \) blocks in the latter category, there are at least \((N + 2 + X) - N = X + 2\) blocks in the former category, and so:

\[
T \geq (X + 2) \times \text{BlockSize} \tag{11.3}
\]

But this contradicts 11.2 above.

**Lemma 11.27.** For any history of \( S \) where Condition 11.10 holds, the elements of \( \text{NextBlock}[1..N] \) are distinct.

**Proof.** The elements of \( \text{NextBlock}[1..N] \) are distinct before and after each call to \( \text{RecycleBlock()} \) by Lemmas 11.23 and 11.26 and Condition 11.10 (a). The distinctness property can only be violated when some element of \( \text{NextBlock}[1..N] \) is written at line 361 of \( \text{RecycleBlock()} \), in which case the state of \( \text{NextBlock}[1..N] \) just after it is written contradicts Lemmas 11.23 and 11.26 and Condition 11.10 (a).

**Lemma 11.28.** For any history \( H \) of \( S \) where Conditions 11.9–11.11 hold, and for any block \( x \), letting \( P(x) \) denote the predicate

\[
\text{block } x \text{ is one of the elements of } \text{FreshBlocks} \text{ or } \text{DirtyBlocks}, \text{ or an element that has been dequeued at line 362 from } \text{RecentBlocks} \text{ and not yet enqueued (into } \text{FreshBlocks} \text{ or } \text{DirtyBlocks}) \text{ at line 366 or line 368, or an element that has been dequeued from } \text{DirtyBlocks} \text{ at line 377 and not yet enqueued into } \text{FreshBlocks} \text{ at line 378}
\]

the following holds throughout \( H \): \( P(x) \) implies \( \text{ref}(x) = 0 \).

**Proof.** Since the property under consideration is a safety property it suffices to consider finite \( H \). We proceed by induction on \( |H| \) and consider the contrapositive of the lemma: \( \text{ref}(x) > 0 \) implies \( \neg P(x) \).
Basis: \(|H| = 0\). At initialization, the initial block is the only block \(x\) for which \(\text{ref}(x) > 0\). Since this block is initially in \(\text{RecentBlocks}\) and not in \(\text{FreshBlocks}\) or \(\text{DirtyBlocks}\), and since every process is in the initial state (i.e., not executing inside \(\text{RecycleBlock}\)), \(\neg P(x)\) holds, as wanted.

Induction step: For any \(k \in \mathbb{N}\), suppose that the lemma holds when \(|H| = k\), and consider \(|H| = k + 1\). Let \(p\) be the process that executes step \(k + 1\). It suffices to consider cases where step \(k + 1\) increases \(\text{ref}(x)\) from zero to a positive value, or causes \(P(x)\) to hold.

Case A: step \(k + 1\) increases \(\text{ref}(x)\) for some block \(x\) from zero to a positive value. We must show that \(\neg P(x)\) holds in state \(H[k + 1]\). It follows from Condition 11.11 (a) that in step \(k + 1\), \(\text{AllocBlock}\) returns \(x\) to \(p\). In this case, \(x\) is the value that \(p\) reads from \(\text{NextBlock}[p]\) in the same step, and so \(x = \text{NextBlock}[p]\) in state \(H[k + 1]\). Now suppose for contradiction that \(P(x)\) does hold in state \(H[k + 1]\).

Subcase A-i: in state \(H[k + 1]\) \(x\) is in \(\text{FreshBlocks}\) or \(\text{DirtyBlocks}\). This contradicts Lemmas 11.23 and 11.26 (distinctness property) because \(x = \text{NextBlock}[p]\) in state \(H[k + 1]\).

Subcase A-ii: in state \(H[k + 1]\) \(x\) has been dequeued by some process \(q\) at line 362 and not yet enqueued at line 366 or line 368. In this case \(q \neq p\), and so we can obtain a state where \(x = \text{NextBlock}[p]\) and also \(x\) is in \(\text{FreshBlocks}\) or \(\text{DirtyBlocks}\) by allowing \(q\) to take a bounded number of additional steps (up to and including line 366 or line 368). That state contradicts Lemmas 11.23 and 11.26 as in Subcase A-i.

Subcase A-iii: in state \(H[k + 1]\) \(x\) has been dequeued by some process \(q\) at line 377 and not yet enqueued \(x\) at line 378. In this case \(q \neq p\), and so we can obtain a state where \(x = \text{NextBlock}[p]\) and also \(x\) is in \(\text{FreshBlocks}\) by allowing \(q\) to take a bounded number of additional steps (up to and including line 378). That state contradicts Lemmas 11.23 and 11.26 as in Subcase A-i.

Case B: in step \(k + 1\) process \(p\) dequeues \(x\) from \(\text{RecentBlocks}\) at line 362. In this case \(P(x)\) holds, and so we must show that \(\text{ref}(x) = 0\) in state \(H[k + 1]\). Suppose that step \(k + 1\) occurs during the \(i\)'th call to \(\text{RecycleBlock}\) (this is well-defined by Condition 11.10 (a)).

Subcase B-i: \(i < N + 1\). In this case \(x\) is one of the blocks that is initially in \(\text{RecentBlocks}\), and is not the initial block, and so \(\text{ref}(x) = 0\) at initialization. Furthermore, it follows from the algorithm that \(x\) is in \(\text{RecentBlocks}\) and not in \(\text{NextBlock}[1..N]\) from \(H[0]\) up to and including \(H[k + 1]\). Consequently, no call to \(\text{AllocBlock}\) returns
x between the initial state and \( H[k+1] \), and so by Condition 11.11 (a) \( \text{ref}(x) \) does not increase above zero before \( H[k+1] \). Thus, \( \text{ref}(x) = 0 \) in state \( H[k+1] \).

**Subcase B-ii: \( i \geq N + 1 \).** Let \( b_j \) denote the \( j \)'th block that becomes current (initial block for \( j = 0 \)) and note that \( x = b_{i-N-1} \). To see this, first note that \( \text{RecentBlocks} \) is accessed as a FIFO queue at lines 360 and 362 of \( \text{RecycleBlock()} \), which has size \( N+1 \) between calls to \( \text{RecycleBlock()} \) by Lemma 11.24. Thus, the element dequeued from \( \text{RecentBlocks} \) (at line 362) during the \( i \)'th call to \( \text{RecycleBlock()} \) is the initial block if \( i = N + 1 \) (since it is at the tail of \( \text{RecentBlocks} \) initially) or else the block that was enqueued during the \((i-N-1)\)'th call to \( \text{RecycleBlock()} \) (at line 360). Furthermore, in the latter case (i.e., \( i > N + 1 \)) \( x \) is the block made current by the process that makes the \((i-N-1)\)'th call to \( \text{RecycleBlock()} \) by the algorithm (line 359 of \( \text{AllocBlock()} \) and line 360 of \( \text{RecycleBlock()} \)) as well as Condition 11.10 (b)–(c).

Now let \( s \) be the initial state if \( i = N + 1 \), or otherwise the state just after the \((i-N-1)\)'th call to \( \text{RecycleBlock()} \) ends. As explained earlier, since \( \text{RecentBlocks} \) is accessed as a FIFO queue of size \( N+1 \) (where size is defined between calls to \( \text{RecycleBlock()} \)), \( x \) is the tail element of \( \text{RecentBlocks} \) in state \( s \). Moreover, \( x \) remains in \( \text{RecentBlocks} \) from state \( s \) until \( p \) dequeues it from \( \text{RecentBlocks} \) in step \( k + 1 \). We will now use this fact to show that \( \text{ref}(x) = 0 \) in state \( H[k+1] \).

First, note that since \( x \) is the \((i-N-1)\)'th block that became current, it follows from Condition 11.11 (b) that \( \text{ref}(x) \) reaches zero after \( x \) becomes current and before the \( i \)'th call to \( \text{RecycleBlock()} \) is made. In particular, this happens after state \( s \), because \( \text{ref}(x) > 0 \) from the moment \( x \) becomes current until state \( s \). (For \( i = N + 1 \) this follows trivially because \( s \) is the initial state, and for \( i > N + 1 \) it follows because the process that makes \( x \) current holds a weak reference to \( x \) until it completes the \((i-N-1)\)'th call to \( \text{RecycleBlock()} \), which is when state \( s \) occurs.) Next, note that \( x \) is not in \( \text{NextBlock}[1..N] \) after state \( s \). For the subhistory after state \( s \) up to and including state \( H[k] \) this follows from Lemmas 11.23 and 11.26 because \( x \) is in \( \text{RecentBlocks} \). For state \( H[k+1] \) this follows from the last statement and from the action of step \( k + 1 \), which does not write \( \text{NextBlock}[1..N] \). Now since \( x \) is not in \( \text{NextBlock}[1..N] \) after state \( s \), no call to \( \text{AllocBlock()} \) returns \( x \) after state \( s \) by the algorithm for \( \text{AllocBlock()} \), and so by Condition 11.11 (a) \( \text{ref}(x) \) does not increase above zero once it has reached zero after state \( s \). Thus, \( \text{ref}(x) = 0 \) in state \( H[k+1] \), as wanted.

**Case C:** in step \( k + 1 \) process \( p \) enqueues \( x \) into \( \text{FreshBlocks} \) at line 366 or \( \text{DirtyBlocks} \) at line 368. Suppose for contradiction that \( \text{ref}(x) > 0 \) in state \( H[k+1] \). Then \( \text{ref}(x) > 0 \)
Chapter 11. Bounding Space Complexity

holds in state $H[k]$ as well by Condition 11.11 (a) and the action of step $k + 1$ (which is not a return from AllocBlock()). Furthermore, in state $H[k]$ $p$ has already dequeued $x$ at line 362 and not yet enqueued at line 366 or line 368, and so $P(x)$ holds. Since $\text{ref}(x) > 0$ and $P(x)$ hold in state $H[k]$, this contradicts the induction hypothesis.

**Case D:** in step $k + 1$ process $p$ dequeues $x$ from DirtyBlocks at line 377. Suppose for contradiction that $\text{ref}(x) > 0$ in state $H[k + 1]$. Then $\text{ref}(x) > 0$ holds in state $H[k]$ as in Case C. Furthermore, in state $H[k]$ $p$ has already dequeued $x$ from DirtyBlocks at line 377 and not yet enqueued it into FreshBlocks at line 378 and so $P(x)$ holds. We arrive at a contradiction as in Case C.

**Case E:** in step $k + 1$ process $p$ enqueues $x$ into FreshBlocks at line 378. Suppose for contradiction that $\text{ref}(x) > 0$ in state $H[k + 1]$. Then $\text{ref}(x) > 0$ holds in state $H[k]$ as in Case C. Furthermore, in state $H[k]$ $p$ has already dequeued $x$ from DirtyBlocks at line 377 and not yet enqueued it into FreshBlocks at line 378 and so $P(x)$ holds. We arrive at a contradiction as in Case C.

**Lemma 11.29.** For any history of $S$ where Conditions 11.9–11.11 hold and for any block $x$, if $x$ is in FreshBlocks then $x$ is fresh.

*Proof.* Since the property under consideration is a safety property it suffices to consider finite $H$. We proceed by induction on $|H|$, noting that every element of FreshBlocks is indeed a block address by Lemmas 11.23 and 11.26.

**Basis:** $|H| = 0$. At initialization, every block that is in FreshBlocks is fresh.

**Induction step:** For any $k \in \mathbb{N}$, suppose that the lemma holds when $|H| = k$, and consider $|H| = k + 1$. Let $p$ be the process that executes step $k + 1$. It suffices to consider cases where step $k + 1$ enqueues some block $x$ into FreshBlocks or else accesses block $x$.

**Case A:** in step $k + 1$ $p$ enqueues $x$ into FreshBlocks at line 366. We must show that $x$ is fresh in state $H[k + 1]$. If $p$ adds $x$ to FreshBlocks at line 366 then $\text{ref}(x) = 0$ holds in state during $p$'s entire execution of lines 363–366 by Lemma 11.28. Consequently, from the moment $p$ begins to soft-reset $x$ at lines 364–365 until state $H[k + 1]$, no process other than $p$ accesses $x$ outside of calls to AllocBlock() and RecycleBlock() by Condition 11.9 (a), or inside such calls by the algorithm for AllocBlock() and by Condition 11.10 (a). Thus, after $p$ soft-resets $x$ at lines 364–365, $x$ remains fresh up to and including state $H[k + 1]$, as wanted.

**Case B:** in step $k + 1$ $p$ enqueues $x$ into FreshBlocks at line 378. If $p$ adds $x$ to FreshBlocks at line 378 then $x$ was the head element of DirtyBlocks during $p$'s execution of lines 370–377 by the algorithm for RecycleBlock() and Condition 11.10 (a), and so
ref(x) = 0 in that part of the history by Lemma 11.28. In this case, note that x was the head element of DirtyBlocks during the last BlockSize executions of lines 370–373. This follows from the success of the test at line 374, from the fact that elements are enqueued into DirtyBlocks only at the tail (line 368), from the initialization of numRegsReset to zero, from the way numRegsReset is incremented at line 381 and reset at line 379, and from Condition 11.10 (a). Thus, the registers in block x were reset one by one over the last BlockSize executions of line 373, following which x ⊲ B and x ⊲ blockVersion were reset at lines 375–376. Now let H’ be the suffix of H starting with the first such execution of line 373. The BlockSize executions of line 373 in H’ guarantee that x is fresh in state H[k + 1] provided that no other access to x occurs during H’, which we show in the remainder of Case B.

Since we showed that ref(x) = 0 holds in H’, it follows by Condition 11.9 (a) that during H’ no process accesses x outside of calls to AllocBlock() and RecycleBlock(). Next, consider possible accesses to x inside such calls. No block is accessed inside AllocBlock(), but a block may be accessed inside RecycleBlock() at lines 364, 365, 373, 375 or 376. All executions of lines 373 and 376 during H’ reset registers in x to their initial values, and have already been considered above. Similarly, all executions of lines 364 and 375 reset x ⊲ B (in the writable implementation) to its initial state. Finally, if some process q accesses x at line 365 in H’ then by allowing q to complete line 366 we obtain a history at the end of which x is in both DirtyBlocks and FreshBlocks, which contradicts Lemmas 11.23 and 11.26 (distinctness property). Thus, x is fresh in state H[k + 1].

Case C: in step k + 1 p accesses block x. We must show that x is not in FreshBlocks in state H[k + 1] or that x is fresh in state H[k + 1]. Since ref(x) = 0 while x is in FreshBlocks by Lemma 11.28, no process may access x outside of calls to AllocBlock() or RecycleBlock() while x is in FreshBlocks by Condition 11.9 (a). On the other hand, if some process q accesses x inside such a call then this occurs at lines 364, 365, 373, 375 or 376. If q does so at lines 364 or 365 then by allowing q to complete line 366 we obtain a state where there are two copies of x in FreshBlocks, which contradicts Lemmas 11.23 and 11.26 (distinctness property). If q does so at lines 373, 375 or 376 then just after this step x is in both FreshBlocks and DirtyBlocks, which again contradicts Lemmas 11.23 and 11.26.

Lemma 11.30. For any history of S where Conditions 11.9–11.11 hold and for any
block $x$, any access to $x$ that occurs inside \texttt{RecycleBlock()} or \texttt{AllocBlock()} occurs while $\text{ref}(x) = 0$ and $x$ is not an element of $\text{NextBlock}[1..N]$.

\textit{Proof.} An access to a block inside \texttt{RecycleBlock()} or \texttt{AllocBlock()} can only occur at lines 364, 365, 373, 375 or 376 of \texttt{RecycleBlock()}. (Note that line 359 of \texttt{AllocBlock()} does not access any block; it only reads a block address.) Suppose for contradiction that such an access occurs by some process $p$ in the last step of a history $H$ while $\text{ref}(x) > 0$ or $x$ is in $\text{NextBlock}[1..N]$.

\textbf{Case A:} the access occurs at lines 364 or 365. By scheduling another step of $p$ whereby it completes line 366, we obtain a history where $x$ is in $\text{FreshBlocks}$. If $\text{ref}(x) > 0$ at that point, this contradicts Lemma 11.28. If $x$ is also in $\text{NextBlock}[1..N]$ at that point, this contradicts Lemmas 11.23 and 11.26.

\textbf{Case B:} the access occurs at lines 373, 375 or 376. It follows from Condition 11.10 (a), $p$'s prior execution of line 371 and the fact that the state of $\text{DirtyBlocks}$ is not changed outside of \texttt{RecycleBlock()} that $x$ is the head element of $\text{DirtyBlocks}$ at the end of $H$. If $\text{ref}(x) > 0$ at that point, this contradicts Lemma 11.28. If $x$ is also in $\text{NextBlock}[1..N]$ at that point, this contradicts Lemmas 11.23 and 11.26.

\textbf{Lemma 11.31.} For any history of $S$ where Conditions 11.9–11.11 hold, if a call to \texttt{AllocBlock()} returns $x$ then $x$ is the address of a block such that $\text{ref}(x) = 1$ and $x$ is fresh.

\textit{Proof.} Since the correctness properties under consideration are safety properties it suffices to consider finite $H$. We fix a process $p$ and proceed by induction on the number $c$ of calls $p$ has made to \texttt{AllocBlock()}. Suppose that the last of these calls returns $x$.

\textbf{Basis:} $c = 1$. The first time a process $p$ calls \texttt{AllocBlock()}, the response $x$ is the initial value of $\text{NextBlock}[p]$, which is the address of a block. It follows from the initialization of $\text{NextBlock}[p]$ that $x$ is not the initial block. Furthermore, by Lemma 11.27 and the algorithm for \texttt{AllocBlock()}, block $x$ is different from any other block returned previously by calls \texttt{AllocBlock()}. Consequently, $\text{ref}(x) = 0$ until $x$ is returned, and so $\text{ref}(x) = 1$ just after $x$ is returned. Block $x$ is also fresh because no process accesses $x$ outside of \texttt{RecycleBlock()} and \texttt{AllocBlock()} while $\text{ref}(x) = 0$ by Condition 11.9 (a), and no process accesses $x$ inside \texttt{AllocBlock()} or \texttt{RecycleBlock()} while $\text{NextBlock}[p] = x$ by Lemma 11.30.

\textbf{Induction step:} for any $k > 0$, suppose that the lemma holds for $p$'s first $k$ calls to \texttt{AllocBlock()}, and consider $p$'s $k + 1$'th call.
Case A: p’s k’th call was invisible. In this case, p’s k’th call to AllocBlock() also returned x; by the induction hypothesis x is the address of a block and at that point it was fresh and ref(x) = 0 held. When p releases its weak reference to x following the k’th call to AllocBlock(), x is still fresh and ref(x) = 0 holds by Condition 11.9 (b). Between p’s k’th and k + 1’th call to RecycleBlock(), NextBlock[p] = x by Condition 11.10 (b) and the fact that no other process overwrites NextBlock[p]. Furthermore, no other element of NextBlock[1..N] contains x by Lemmas 11.23 and 11.26. Thus, no call to AllocBlock() by another process returns x, and so by Conditions 11.9 (a) and 11.11 (a), ref(x) does not increase above zero and no other process accesses x outside of RecycleBlock() and AllocBlock(). Similarly, while NextBlock[p] = x no process accesses x inside AllocBlock() or RecycleBlock() by Lemma 11.30. This implies that x is still fresh and ref(x) = 0 when p’s k + 1’th call to AllocBlock() returns x.

Case B: p’s k’th call was visible. In this case, p executes RecycleBlock() between its k’th and k + 1’th call to AllocBlock() by Condition 11.10 (b). During this call to RecycleBlock(), p overwrites NextBlock[p] with an element y dequeued from FreshBlocks at line 361. When p dequeues y from FreshBlocks, it follows from Lemmas 11.23, 11.26 and 11.29 that y is the address of a block that is fresh, and it follows from Lemma 11.28 that ref(y) = 0. It also follows from Lemmas 11.23 and 11.26 that no element of NextBlock[1..N] stores y’s address just before p dequeues y from FreshBlocks. By Conditions 11.9 (a) and 11.11 (a) and the algorithm for AllocBlock(), ref(y) does not increase and no process accesses y outside of AllocBlock() and RecycleBlock() between p removing y from FreshBlocks and p writing y to NextBlock[p] at line 361. Furthermore, by Condition 11.10 (a) no process accesses y inside RecycleBlock() or AllocBlock() between these two steps. Thus, y is fresh and ref(y) = 0 when p writes y to NextBlock[p] at line 361.

After p writes y to NextBlock[p] at line 361 and before p’s k+1’th call to AllocBlock() returns y, y remains fresh and ref(y) = 0 continues to hold. For ref(y) = 0, this is because NextBlock[p] = y, by the distinctness of the addresses in NextBlock[1..N] (Lemmas 11.23 and 11.26), by the algorithm for AllocBlock() and by Condition 11.11 (a). For y being fresh, this is because no process accesses y outside of calls to RecycleBlock() and AllocBlock() since ref(y) = 0 and Condition 11.9 (a) holds, and no process accesses y inside RecycleBlock() or AllocBlock() while NextBlock[p] = y by Lemma 11.30. Thus, when p’s k + 1’th call to AllocBlock() returns y, y is fresh and ref(y) = 1 since p holds a weak reference to y.
Theorem 11.32. For any history of $S$ where Conditions 11.9–11.11 hold, Specifications 11.7–11.8 also hold and every call to $\text{AllocBlock()}$ and $\text{RecycleBlock()}$ incurs $O(1)$ RMRs.

Proof. Specification 11.7 (a) follows directly from Lemma 11.31. Specification 11.7 (b) follows directly from Lemma 11.30. Specification 11.8 and $O(1)$ RMR complexity of $\text{AllocBlock()}$ and $\text{RecycleBlock()}$ follow directly from the structure of these functions. For RMR complexity, note that the queues $\text{FreshBlocks}$, $\text{DirtyBlocks}$ and $\text{RecentBlocks}$ can be implemented easily using read/write registers at a cost of $O(1)$ RMRs, as mentioned at the beginning of Section 11.3.

Theorem 11.33. The block allocator presented in this section uses $3N + 5$ blocks.

Proof. It suffices to count the number of blocks at initialization: $N + 4$ are in $\text{FreshBlocks}$, $\text{DirtyBlocks}$ is empty, $N + 1$ are in $\text{RecentBlocks}$ and $N$ more are in $\text{NextBlock}[1..N]$. The total is $3N + 5$, as wanted.

11.4 Conclusion

In this chapter we showed how to bound space complexity for the $O(1)$-RMR implementations presented in Chapters 6, 8 and 10. This required that we instrument our implementations with special subroutines (see Section 11.1) and construct a block allocator (see Section 11.3). We also proposed a special technique for resetting blocks efficiently (see Section 11.2).

Our block allocator uses $O(N)$ blocks (see Theorem 11.33). In the ECAS implementation from Chapter 6 a single allocator is used and each block requires $O(N^2 \log N)$ space, which is dominated by the space complexity of the leader election algorithm [11] that underlies our name consensus and pseudo-lock. Thus, the space complexity of our implementation is $O(N^3 \log N)$. On the other hand, in the writable ECAS implementation from Chapter 8 there are two “layers” of blocks. There are $O(N)$ “top-level” blocks used by the access procedures presented in Figure 8.2. Each of these blocks in turn contains a non-writable ECAS object $B$ that internally uses its own block allocator as in Chapter 6. Thus, the writable ECAS implementation has space complexity $O(N^4 \log N)$.

Our memory management techniques must be adapted in order to work with locally-accessible implementations, otherwise they break the locality properties. We deal with
this issue next in Chapter 12. We then discuss how our approach to memory management can be simplified in Chapter 13.
Chapter 12

Bounding Space Complexity in Locally-Accessible ECAS Implementations

In this chapter we explain why the memory management scheme from Chapter 11 breaks the locality properties when applied to the locally-accessible implementations from Chapter 7 and Section 8.2. We then discuss how to modify this scheme to ensure locality in each of the three shared memory models under consideration: write-back and write-through cache-coherent, and distributed shared memory.

12.1 Write-Back CC Model

Let $I_E^*$ be our non-writable locally-accessible ECAS implementation for the write-back CC model, exactly as described in Section 7.2. Let $I_E^{**}$ be $I_E^*$ transformed by applying the memory management techniques described in earlier sections of Chapter 11. (This transformation instruments $I_E^*$ as described in Section 11.1 and replaces AllocBlock() with the block allocator described in Sections 11.2–11.3.) Similarly, let $I_{EW}^*$ be our writable locally-accessible ECAS implementation for the write-back CC model, as described in Chapter 8 but with the base object $B$ in each block satisfying bounded space complexity in addition to the properties stated in Chapter 8: linearizability, termination, $O(1)$ RMR complexity and locality in the write-back CC model. Let $I_{EW}^{**}$ be $I_{EW}^*$ transformed by applying the memory management techniques described in earlier sections of Chapter 11. (Although we have not presented an implementation up to this point that is suitable for
the base object $B$ in $I_{E}^{**}$ and $I_{EW}^{**}$, we will show in this chapter that $I_{E}^{**}$ has the required properties. In particular, the locality of $I_{E}^{**}$ follows from Theorem 12.1 below.)

Since we have already established the linearizability, termination and $O(1)$-RMR complexity properties of $I_{E}^{**}$ and $I_{EW}^{**}$ in Theorem 11.12, it suffices to show that $I_{E}^{**}$ and $I_{EW}^{**}$ also satisfy the locality property in the write-back CC model without any further transformations.

**Theorem 12.1.** The implementations $I_{E}^{**}$ and $I_{EW}^{**}$ satisfy the locality property in the write-back CC model (see Definition 7.3).

**Proof.** Consider any history $H$ of $I_{E}^{**}$ or $I_{EW}^{**}$, and consider the linearization $\bar{H}$ of that history as defined in our proof of linearizability (for $I_{E}^{**}$ see Lemmas 6.15 and 7.11; for $I_{EW}^{**}$ see Lemmas 8.5 and 8.12).

**Property (R):** Fix process $p$ and a contiguous subsequence $\bar{H}'$ of $\bar{H}$ containing read-like operation executions only. Let $H'$ denote the subsequence of base object atomic steps in $H$ corresponding to $\bar{H}'$. We must show that $p$ incurs at most $O(1)$ RMRs in $H$ performing its steps in $H'$. For steps executed outside of the subroutines BeforeOp, AfterOp and MiddleOp this follows from property (R) of $I_{E}$ because these subroutines do not access any shared variables used by $I_{E}$, and so their execution does not affect the RMR cost of accessing such variables elsewhere. It remains to consider the cost of executing the subroutines themselves.

It follows that $p$ does not execute MiddleOp at all during $H'$, since that only happens during a successful ECAS operation execution, or a Write operation execution in the case of $I_{EW}^{**}$, both of which are write-like and would appear in $\bar{H}'$. (If a process calls MiddleOp during some operation execution $Op$ in $H$, the timestamp of $Op$ is defined and a counterpart of $Op$ appears in $\bar{H}$. ) Now consider BeforeOp and AfterOp. Here $p$ may only read $WaiterID[p]$, write $Active[p]$ and write $WaitFlag[1..N][p]$. If no other process accesses these variables in $H$ between $p$’s first and last step in $H'$, then $p$ incurs one RMR accessing each variable in the write-back CC model. Otherwise, each access to these variables by another process may cause at most one additional RMR for $p$ (by either invalidating $p$’s cached copy of the variable, or downgrading its status from “exclusive” to “shared”). Thus, it suffices to show that there are $O(1)$ such RMRs.

The only place where a process other than $p$ may access $WaiterID[p]$, $Active[p]$ or $WaitFlag[1..N][p]$ is at lines 336–348 of MiddleOp. It follows from Lemma 11.15 that these lines are executed in mutual exclusion and that every $N$’th execution of these lines
accesses the array elements at index $p$. At lines 339–341 all three arrays are accessed and each access may cause at most one additional RMR for $p$ (by invalidating $p$’s cached copy). The only other possible access occurs in the busy-wait loop at line 343, where a process $q$ reads $\text{WaitFlag}[q][p]$ repeatedly until it reads $\text{true}$. This causes at most two RMRs for $p$ (by downgrading $p$’s exclusive cached copy of $\text{WaitFlag}[q][p]$ to a shared copy) because in the first such RMR $p$ writes $\text{true}$ to $\text{WaitFlag}[q][p]$ at line 328 of $\text{AfterOp}$, which causes $q$’s busy-wait loop at line 343 to terminate after only one more read of $\text{WaitFlag}[q][p]$, which in turn causes at most one additional RMR for $p$. (Note that no process resets $\text{WaitFlag}[q][p]$ to $\text{false}$ while $q$ is at line 343 by the algorithm and Lemma 11.15.) Thus, a single execution of lines 336–348 causes $O(1)$ additional RMRs for $p$ if $c = p$ holds, and zero additional RMRs otherwise.

To complete the proof of property (R), it suffices to show that there are $O(1)$ executions (partial or complete) of lines 336–348 of $\text{MiddleOp}$ where $c = p$ in $H$ between $p$’s first and last step in $H'$. It suffices to consider the subhistory $H''$ of $H$ after the first step of $p$’s second operation execution in $\bar{H}'$ and before the last step of $p$’s second-last operation execution in $\bar{H}'$ (because we know that $p$ incurs $O(1)$ RMRs in $H$ executing its first and last operation executions in $\bar{H}'$). To that end, suppose that there are two or more (partial or complete) executions of lines 336–348 of $\text{MiddleOp}$ where $c = p$ in $H''$. Then Lemma 11.15 implies that at least $N + 1$ (partial or complete) executions of $\text{MiddleOp}$ occur in $H''$, and so some process $z$ performs two of these. This implies that $z$ applies two successful operation executions of type $\text{Write}$ or $\text{ECAS}$ in $\bar{H}$ corresponding to these calls to $\text{MiddleOp}$. Let $Op$ denote $z$’s second operation execution and note that $z$’s successful $\text{changeCurBlock}$ during $Op$ occurs in $H''$. Thus, it follows from our construction of $\bar{H}$ and definition of $H''$ that $Op$ is linearized after $p$’s first operation execution in $\bar{H}'$ and before $p$’s last operation execution in $\bar{H}'$, which contradicts $\bar{H}'$ consisting of read-like operation executions only.

**Property (W):** Fix process $p$ and a contiguous subsequence $\bar{H}'$ of $\bar{H}$ containing operation executions by $p$ only. Again let $H'$ be the corresponding subsequence of base object atomic steps in $H$. As in the proof of (R) it suffices to show that $p$ incurs $O(1)$ RMRs executing steps corresponding to $\text{BeforeOp}$, $\text{AfterOp}$ and $\text{MiddleOp}$ because $p$’s remaining steps incur $O(1)$ RMRs in $H$ by property (W) of $I^*_E$. To that end, we proceed as in the proof of property (R), and derive a similar contradiction. Define $H''$ as before and suppose that there are two or more executions of lines 336–348 of $\text{MiddleOp}$ in $H''$ where $c = p$. Then some process $z \neq p$ applies a successful $\text{ECAS}$ operation execution, or
a \textbf{Write} operation execution in the case of $I_{EW}^{**}$, in $\bar{H}$ that is linearized between $p$'s first and last operation execution in $\bar{H}'$, which contradicts $\bar{H}'$ being a contiguous subsequence of $\bar{H}$ containing operation executions by $p$ only. (In this case $z \neq p$ follows from the test $c \neq \text{PID}$ at line 338 of MiddleOp.)

\section*{12.2 Write-Through CC Model}

Define $I_{E}^{*}$, $I_{E}^{**}$, $I_{EW}^{*}$ and $I_{EW}^{**}$ as in Section 12.1. On first impression it may seem that $I_{E}^{**}$ and $I_{EW}^{**}$ should satisfy the locality property in the write-through CC model (see Definition 7.2) because they satisfy the locality property in the write-back CC model (see Definition 7.3) and because the former property is weaker than the latter. However, this is not the case because the locality property is not the only thing that has changed between Section 12.1 and this section: the shared memory model itself is also weaker. That is, for any history $H$ of $I_{E}^{**}$ or $I_{EW}^{**}$, the number of RMRs incurred in $H$ in the write-through CC model is at least the number incurred in the write-back CC model, and for some $H$ the number of RMRs is higher in the write-through CC model. Thus, locality of an implementation in the write-back CC model does not automatically imply locality in the write-through CC model.

Although the implementation $I_{E}^{*}$ satisfies the locality property in the write-through and write-back CC models, transforming it to $I_{E}^{**}$ by applying memory management breaks the locality property in the write-through CC model. This is because every process executes the subroutines \texttt{BeforeOp()} and \texttt{AfterOp()} (see Figures 11.2–11.3) at the beginning and end of any operation execution on the target object, and inside these functions each process writes a shared register (i.e., $\text{Active}[\text{PID}]$). Thus, each operation execution on the target object incurs $\Omega(1)$ RMRs in the write-through CC model, while the locality property (Definition 7.2) requires that in some cases an entire sequence of operation executions incur only $O(1)$ RMRs in total.

To remedy the problem described above, we allow processes to apply certain operation executions without executing \texttt{BeforeOp()} or \texttt{AfterOp()}. This is challenging because bypassing these subroutines creates the possibility that a process accesses a block (outside of \texttt{AllocBlock()} or the accessory function \texttt{RecycleBlock()}) while that block is being recycled. We therefore break down the presentation and analysis of the solution into several steps. First, we change the ECAS type slightly by introducing a new operation type called \texttt{hasChanged()}, and we describe how this operation type is implemented. We
call the new type $\tau_{ECAS+}$, and we call its writable counterpart $\tau_{ECAS-W+}$. Then, we show that in certain cases calls to \texttt{hasChanged()} can be applied “in-cache”. Finally, we use this property to construct a locally-accessible ECAS implementation for the write-through CC model.

The new operation type we add to the ECAS type \texttt{hasChanged()} has no effect on the state of the target object, but rather returns information that can be used to detect state changes. Recall that this state consists of a value $V$ and a Boolean array $\text{Linked}[1..N]$. The response is a Boolean and to define it formally we introduce a third component into the state definition – a Boolean array $\text{Crumb}[1..N]$, initially false. A \texttt{hasChanged()} operation by $p$ returns the negation of $\text{Crumb}[p]$ and leaves the state unchanged. An operation of any other type by $p$ works as before (see Figures 6.1, 6.2 and 6.3) except that it sets $\text{Crumb}[p]$ to true and, if it is write-like, it resets all the other entries of $\text{Crumb}[1..N]$ to false. (For types $\tau_{ECAS+}$ and $\tau_{ECAS-W+}$, we classify \texttt{Read}, \texttt{LL}, \texttt{ECAS} and \texttt{Write} operations as either read-like or write-like as described in Chapter 7, and we classify all \texttt{hasChanged()} operations as read-like. This means that some read-like operations write $\text{Crumb}[1..N]$, but as explained in Chapter 7 we are free to apply such a classification.)

The semantics of \texttt{hasChanged()} are best illustrated in the context of a sequential history $H$ over an object of type $\tau_{ECAS+}$ or $\tau_{ECAS-W+}$. Fix a process $p$, let $Op$ be a \texttt{hasChanged()} operation by $p$ in $H$, and let $Op'$ be the last non-\texttt{hasChanged()} operation by $p$ that precedes $Op$ in $H$. If $Op'$ is undefined, then $Op$ returns true. Otherwise, letting $H'$ denote the subhistory of $H$ consisting of the operations that follow $Op'$ and precede $Op$, $Op$ returns true if $H'$ contains a write-like operation (necessarily by another process), and false otherwise. Example of this are shown in Figure 12.1.

We create $O(1)$-RMR linearizable implementations of $\tau_{ECAS+}$ and $\tau_{ECAS-W+}$, denoted $I_{E+}$ and $I_{EW+}$ respectively, by modifying our implementations of $\tau_{ECAS}$ and $\tau_{ECAS-W}$ with bounded space complexity. Let $I_E$ and $I_{EW}$ denote the implementations of ECAS and writable ECAS from Chapters 6 and 8, respectively. To obtain $I_{E+}$ and $I_{EW+}$ from $I_E$ and $I_{EW}$, we apply transformation that changes the structure of blocks and introduces an access procedure for operation type \texttt{hasChanged()}. We then bound the space complexity of $I_{E+}$ and $I_{EW+}$, obtaining implementations $I_{EB+}$ and $I_{EBW+}$ of types $\tau_{ECAS+}$ and $\tau_{ECAS-W+}$, respectively. We describe the construction of $I_{EB+}$ and $I_{EBW+}$ in Section 12.2.1.

Our linearizable implementations $I_{EB+}$ and $I_{EBW+}$ satisfy $O(1)$ RMR complexity per
Chapter 12. Space Complexity in Locally-Accessible ECAS

$H_1$: (INV, $p, v, \text{hasChanged}()$), (RES, $p, v, \text{true}$)

$H_2$: (INV, $p, v, \text{Read}()$), (RES, $p, v, 0$) (INV, $p, v, \text{hasChanged}()$), (RES, $p, v, \text{false}$)

$H_3$: (INV, $p, v, \text{Read}()$), (RES, $p, v, 0$) (INV, $q, v, \text{ECAS}(\text{false}, 0, 1)$), (RES, $q, v, (0, \text{true})$) (INV, $p, v, \text{hasChanged}()$), (RES, $p, v, \text{true}$)

Note: In each history operation executions are denoted using tuples in the format defined in Chapter 2.

Figure 12.1: Example histories illustrating the use of $\text{hasChanged}()$.

operation as well an additional RMR complexity property similar in spirit to (but weaker than) the locality property (see Theorem 12.3). For the latter property, we require two things. First, the block manager must not only have $O(1)$-RMR complexity per operation but must also satisfy the locality property for the write-through CC model (see Definition 7.2). This is the same requirement as in Section 7.2, where we discussed a locally-accessible implementation of ECAS without memory recycling. As we showed in Lemma 7.17, the block manager implementation presented in Chapter 5 satisfies this property. Second, in the case of $I_{EW^+}$, the base object $B$ must be implemented using $I_{E^+}$ (or another implementation of $\tau_{ECAS^+}$ that has the same linearizability, termination and RMR complexity properties).

In the remainder of this section we describe and analyze the implementations $I_{E^+}$ and $I_{EW^+}$ (Section 12.2.1), and then we use these implementations to construct locally-accessible implementations of ECAS with bounded space complexity (Section 12.2.2): an implementation $I^{BL}_{E}$ of type $\tau_{ECAS}$ (i.e., non-writable ECAS), and an implementation $I^{BL}_{EW}$ of type $\tau_{ECAS-W}$ (i.e., non-writable ECAS).

12.2.1 Implementations $I^{B}_{E^+}$ and $I^{B}_{EW^+}$

In this section we show how to implement $I^{B}_{E^+}$ and $I^{B}_{EW^+}$, and prove that these implementations are linearizable $O(1)$-RMR implementations of $\tau_{ECAS^+}$ and $\tau_{ECAS-W^+}$, respectively, with bounded space complexity. We also show that these implementations satisfy an additional RMR complexity property, as mentioned earlier, that is similar in spirit to locality but weaker. Since the treatment of $I^{B}_{E^+}$ and $I^{B}_{EW^+}$ is similar, we will
Chapter 12. Space Complexity in Locally-Accessible ECAS

To obtain $I_{E+}$ from $I_E$, we introduce a Boolean array $Crumb[1..N]$ into each block, all elements initialized to false. The state of this array corresponds to the component $Crumb[1..N]$ in the state of type $\tau_{ECAS+}$ just as the array $Linked[1..N]$ in each block corresponds to the component $Linked[1..N]$ in $\tau_{ECAS}$. Each time a process $p$ executes an operation of a type other than $\text{hasChanged}()$, it ensures that $Crumb[p]$ in the current block is true. That is, immediately after calling $\text{getCurBlock}()$, say with response $x$, $p$ writes $x \triangleright Crumb[p] = \text{true}$. (This means that every non-$\text{hasChanged}()$ operation execution incurs at least one RMR, which is acceptable because $I_{E+}$ is only a stepping stone towards a locally-accessible implementation in this section.) To execute $\text{hasChanged}()$, $p$ calls $\text{getCurBlock}()$, say with response $x$, then reads $x \triangleright Crumb[p]$, and returns true if and only if it read is false. Note that the value of $x \triangleright Crumb[p]$ is never reset to false for a particular $x$, but a change of the current block by a process other than $p$ causes the next $\text{hasChanged}()$ by $p$ to return true. Intuitively, this is the desired behaviour because write-like operation executions coincide with changes to the current block. (Recall our classification of operations in Chapter 7.)

To obtain $I_{EW+}$ from $I_{EW}$, we follow an analogous approach. The only difference is in the access procedure for $\text{hasChanged}()$, which must not only check whether the current block has changed, but must in some cases also check for a change in the state of $B$ in the current block by calling $B.\text{hasChanged}()$. This is because in $I_{EW}$, not every write-like operation execution changes the current block; only $\text{Write}()$ operation executions do so. To execute $\text{hasChanged}()$, $p$ calls $\text{getCurBlock}()$, say with response $x$, then reads $x \triangleright Crumb[p]$, and returns true if it read false. However, if $p$ read true, it calls $(x \triangleright B).\text{hasChanged}()$ and returns the response of this call.

Having shown how to create $I_{E+}$ and $I_{EW+}$, we now turn our attention to memory management. (We continue to ignore locality, which is dealt with in another layer later on.) To that end, we construct counterparts $I_{E+}^B$ and $I_{EW+}^B$ of $I_{E+}$ and $I_{EW+}$, respectively by applying the memory management techniques described in Chapter 11 with one exception: in the access procedure for $\text{hasChanged}()$, calls to $\text{BeforeOp}()$ and $\text{AfterOp}()$ are omitted to ensure that a call to $\text{hasChanged}()$ applies read-like base object operations only. This unfortunately raises the possibility that when a process $p$ accesses $x \triangleright Crumb[p]$ or $x \triangleright B$ during a $\text{hasChanged}()$ operation execution, block $x$ has undergone recycling since $p$’s call to $\text{getCurBlock}()$ with response $x$, which violates Specification 11.7 (b). However, as we show in Theorem 12.2, this has no ill side-effects and the response of
hasChanged() is correct. Intuitively, this is because from p’s point of view, recycling of block x can only reset \(x \triangleright \text{Crumb}[p]\) from true to false; it cannot set \(x \triangleright \text{Crumb}[p]\) from false to true. If p reads \(x \triangleright \text{Crumb}[p]\) after it is reset, then p’s hasChanged() returns true and it can be linearized correctly because a write-like operation execution by another process must have occurred since p determined that x is current. (We simply move the default linearization point of p’s hasChanged(), which is tied to p’s getCurBlock() with response x, to some point later in the history.)

**Theorem 12.2.** Implementations \(I_E^B\) and \(I_{EW+}^B\) satisfy Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4, as well as bounded space complexity. Furthermore, each operation execution on the target object in histories of these implementations incurs \(O(1)\) RMRs in the write-through CC model.

**Proof.**

**Specification 6.1:** First, consider \(I_E^+\) and \(I_{EW+}^+\), on which \(I_E^B\) and \(I_{EW+}^B\) are based. The linearizability of \(I_E^+\) and \(I_{EW+}^+\) (under Condition 6.4) follows from the original proofs in Chapters 6 and 8, except that we must apply some additional analysis for a hasChanged() operation execution. (The modifications to the access procedures for other operation types introduce additional base object atomic steps but these only access the block field \(\text{Crumb}[1..N]\), and do not affect the invariants used to prove linearizability.) We also introduce an additional invariant:

Let \(H\) be a history of \(I_E^+\). We construct a candidate linearization \(\bar{H}\) of \(H\) as before, where the timestamp for a hasChanged() operation execution \(Op\) by process p is defined as follows: if p accesses block x and reads \(x \triangleright \text{Crumb}[p]\) during \(Op\) in step t of \(H\) then the timestamp is \((x, t, 0)\). (The timestamp is undefined if \(Op\) is pending.) Since every hasChanged() operation execution is read-like and since the access procedure for hasChanged() does not write any base object, the additional analysis for hasChanged() amounts to showing that the response of \(Op\) is correct in \(\bar{H}\). To that end, note that \(Op\) returns true if and only if p reads false from \(x \triangleright \text{Crumb}[p]\) in step t, which occurs if and only if p never before accessed block x outside of a hasChanged() operation execution, or else some process \(q \neq p\) has changed the current block since p’s last such operation execution \(Op'\). The latter case occurs if and only if a write-like operation execution by q has taken effect between \(Op'\) and \(Op\). Thus, the response of hasChanged() is correct.

For \(I_{EW+}^+\), we follow a similar approach as for \(I_E^+\). Let \(H\) be a history of \(I_{EW+}^+\). The timestamp for a completed hasChanged() operation execution \(Op\) in \(H\) where p accesses
block \( x \) and reads \( \text{false} \) from \( x \triangleright Crumb[p] \) in step \( t \) of \( H \) is \((x, t, 0)\). If \( p \) reads \( \text{true} \) from \( x \triangleright Crumb[p] \) and then accesses \( x \triangleright B \) in step \( t' \) then the timestamp is \((x, t', 0)\). (The timestamp is once again undefined if \( Op \) is pending.) A \textit{hasChanged()} operation execution \( Op \) by \( p \) returns \text{true} if and only if \( p \) reads \text{false} from \( x \triangleright Crumb[p] \) or else receives response \text{true} from \((x \triangleright B)\).\textit{hasChanged()}. Thus, \( Op \) returns \text{true} if and only if \( p \) never before accessed block \( x \) outside of a \textit{hasChanged()} operation execution, or else some process \( q \neq p \) has changed the current block since \( p \)’s last such operation execution \( Op' \), or else some process \( q \neq p \) has applied a write-like atomic step on \( x \triangleright B \) between \( p \)’s atomic steps on this base object in \( Op' \) and \( Op \). The latter two cases occur if and only if a write-like operation execution by \( q \) has taken effect between \( Op' \) and \( Op \). Thus, the response of \textit{hasChanged()} is correct.

It remains to show that \( I_{E+}^B \) and \( I_{EW+}^B \) are linearizable given that \( I_{E+} \) and \( I_{EW+} \) are linearizable (under Condition 6.4). First consider \( I_{E+}^B \), which is the implementation of \( \tau_{ECAS+} \) obtained as discussed above from \( I_{E+} \) using techniques from Chapter 11. Linearizability of \( I_{E+}^B \) follows from Theorem 11.12 with some additional analysis to address the case when a process \( p \) applies a \textit{hasChanged()} operation execution \( Op \) where it calls \textit{getCurBlock()} with response \( x \) and where \( x \) undergoes recycling either before or while \( p \) accesses \( x \triangleright Crumb[p] \). (The responses of operation executions of other types remain correct because \textit{hasChanged()} is read-like and the access procedure for this operation type applies only read-like base object atomic steps.) This scenario is possible because \( p \) does not call \textit{BeforeOp()} during such an operation execution, and so other processes have no way of knowing that \( p \) is about to access block \( x \). If \( p \) does so, then the recycling of block \( x \) may cause \( p \) to read \text{false} from \( x \triangleright Crumb[p] \) even though the value of this variable was \text{true} just after \( p \)'s call to \textit{getCurBlock()}, but it will never cause \( p \) to read \text{true} from \( x \triangleright Crumb[p] \) even though the value of this variable was \text{false} just after \( p \)'s call to \textit{getCurBlock()}. In the former case, we know that block \( x \) ceased to be current before it underwent recycling (see Definition 11.3 and Condition 11.7 (b)), and so some process \( q \neq p \) applied a successful \textit{chgCurBlock}(\( x, y \)) for some block \( y \neq x \) (see Lemma 11.13) after \( p \)'s \textit{getCurBlock()} during \( Op \) and before \( p \) read \( x \triangleright Crumb[p] \). If the latter read occurred in step \( t \) of \( H \), then it suffices to change the timestamp of \( Op \) from \((x, t, 0)\) to \((y, t, 0)\). This ensures that \( Op \) (which returns \text{true}) is linearized after \( q \)'s operation execution, and so there is a write-like operation execution by \( q \) that is linearized before \( Op \) and after \( p \)'s last operation execution where it assigned \( x \triangleright Crumb[p] = \text{true} \).

Now consider \( I_{EW+}^B \), which is the implementation of \( \tau_{ECAS-W+} \) obtained as discussed
above from \(I_{EW+}\) using techniques from Chapter 11. The analysis is the same as for \(I_{E+}\), except that we must consider two more special cases that may arise when \(p\)'s read of \(x \triangleright Crumb[p]\) returns \text{true}. First, \(p\) may call \((x \triangleright B).\text{hasChanged()}\) concurrently with a call to \((x \triangleright B).\text{Reset()}\) by another process in \text{RecycleBlock()} (see lines 364 and 375 in Figure 11.7), which violates an assumption underlying the implementation of the \text{Reset()} operation type (see discussion at the end of Section 11.2). It turns out that this is not a problem because \(p\)'s call to \((x \triangleright B).\text{hasChanged()}\) does not write any base object in the implementation of \(x \triangleright B\), and so it does not interfere with the \text{Reset()} operation. Second, it is possible that \(p\)'s call to \((x \triangleright B).\text{hasChanged()}\) during \(Op\) may return \text{true} even though the same call applied just after \(p\)'s call to \text{getCurBlock()} would have returned \text{false} (but the same cannot occur with \text{true} and \text{false} interchanged). This situation arises when another process applies a \text{Reset()} on \(x \triangleright B\) in \text{RecycleBlock()} after \(p\)'s \text{getCurBlock()} and before \(p\)'s \((x \triangleright B).\text{hasChanged}().\) (Recall that \text{Reset()} changes the current block in the implementation of \(x \triangleright B\).) Once again, to linearize \(Op\) correctly it suffices to change its timestamp from \((x, t, 0)\) to \((y, t, 0)\) where \(t\) is the index of the step in which \(p\) applies \((x \triangleright B).\text{hasChanged}().\) and \(y\) is the next block that becomes current after \(p\)'s \text{getCurBlock()} with response \(x\).

Specification 6.2: Termination for \(I_{E+}\) and \(I_{EW+}\) follows from the termination property of \(I_E\) and \(I_{EW}\), and from the structure of the access procedure for \text{hasChanged}(), which incurs \(O(1)\) steps and applies read-like base object atomic steps only (hence does not affect the proof of termination for other operation types). Termination for \(I_{E+}^B\) and \(I_{EW+}^B\) follows from the termination of \(I_{E+}\) and \(I_{EW+}\) as in the proof of Theorem 11.12, and also from Theorem 11.22.

Bounded space complexity: This follows from the construction of \(I_{E+}^B\) and \(I_{EW+}^B\) from \(I_{E+}\) and \(I_{EW+}\), which applies the memory management techniques described in Chapter 11.

RMR complexity: The analysis is analogous to the analysis above of Specification 6.2.

\begin{proof}

\end{proof}

**Theorem 12.3.** For any history \(H\) of implementation \(I_{E+}^B\) or \(I_{EW+}^B\) and for any process \(p\), let \(\bar{H}\) denote the linearization of \(H\) defined in our proof of linearizability in Theorem 12.2, and let \(\bar{H}'\) denote a contiguous subsequence of read-like operation executions in \(\bar{H}\) such that \(\bar{H}'|p\) contains only \text{hasChanged()} operation executions that return \text{false}. Then process \(p\) incurs \(O(1)\) RMRs in \(H\) applying its operation executions from \(\bar{H}'\).
Chapter 12. Space Complexity in Locally-Accessible ECAS

Proof. Let $H'$ be the subsequence of base object atomic steps in $H$ corresponding to $\bar{H}'$. Since all of $p$'s $\text{hasChanged}()$ operation executions in $\bar{H}'$ return false, and since they are consecutive in $\bar{H}|p$, it follows that in $H'$ process $p$ accesses exactly one block, say $x$, and moreover $x$ remains the current block between $p$'s first and last $\text{getCurBlock}()$ in $H'$ (as a change of the current block can only happen during a write-like operation execution that would appear in $\bar{H}'$). Thus, by the locality property of the block manager in the write-through CC model (see Definition 7.2) $p$'s calls to $\text{getCurBlock}()$ in $H'$ incur $O(1)$ RMRs in $H$. Next, note that no process writes $x \triangleright \text{Crumb}[p]$ between $p$'s first and last step in $H'$, and so $p$ incurs $O(1)$ RMRs accessing this variable. This is because process $p$ does not write $x \triangleright \text{Crumb}[p]$ since $H'|p$ contains $\text{hasChanged}()$ operation executions only, and because a process different from $p$ does not write $x \triangleright \text{Crumb}[p]$ except while hard-resetting block $x$, and not before $p$'s last read of $x \triangleright \text{Crumb}[p]$ in $H'$ (otherwise $p$'s corresponding $\text{hasChanged}()$ returns true). Finally, if $H$ is a history of the writable implementation $I_{EW,+}^B$, we must also consider the cost of $p$'s accesses to $x \triangleright B$ in $H'$. Since $H'$ is a contiguous subsequence of read-like operation executions in $\bar{H}$, it follows that no write-like base object atomic steps are applied to $x \triangleright B$ in $H$ between the first and last step in $H'$, and so by the locality property of $B$ in the write-through CC model, $p$'s accesses to $x \triangleright B$ in $H'$ incur $O(1)$ RMRs in $H$. \qed

12.2.2 Implementations $I_{E}^{BL}$ and $I_{EW}^{BL}$

Implementations $I_{E}^{BL}$ and $I_{EW}^{BL}$ are our locally-accessible implementations of $\tau_{ECAS}$ and $\tau_{ECAS-W}$ respectively, for the write-through CC model, with bounded space complexity. Each implementation uses a single base object $B$ obtained using either $I_{E,+}^B$ (in the case of $I_{E}^{BL}$) or $I_{EW,+}^B$ (in the case of $I_{EW}^{BL}$). This shared object $B$ records the state of the target object. Processes also “cache” pieces of the state of target object using a pair of private variables: for process $p$ variable $\text{lastValue}_p$ records the component $V$ of the state and $\text{lastLinked}_p$ records $\text{Linked}[p]$. We will assume in our analysis that any updates to these private variables occur atomically with the operation on $B$ that precedes them in an access procedure.

The access procedures for $I_{EW}^{BL}$ are presented in Figure 12.2. Implementation $I_{E}^{BL}$ uses the same access procedures, except that the one for $\text{Write}()$ is irrelevant. These access procedures work as follows. Whenever a process $p$ applies a write-like operation execution, it applies the corresponding operation on $B$ and updates $\text{lastValue}_p$ and $\text{lastLinked}_p$ to
match the new state. To apply a read-like operation execution, process \( p \) process first calls \( B\text{.hasChanged()} \) to determine whether \( lastValue_p \) and \( lastLinked_p \) match the current state. If \( B\text{.hasChanged()} \) returns \textit{false} then they do match, and so \( p \) can compute the response of its read-like operation execution using only these private variables. Otherwise \( p \) proceeds as in the case of a write-like operation execution. An ECAS operation execution is a special case because \( p \) does not know beforehand whether it will be read-like or write-like. In this case, \( p \) optimistically calls \( B\text{.hasChanged()} \) at line 398. If the response is \textit{false}, then once again \( p \)'s private variables \( lastValue_p \) and \( lastLinked_p \) match the current state of the target object. Process \( p \) then determines at line 400 whether its ECAS, if applied from the latter state, would be write-like. If not, then \( p \) can compute its response from \( lastValue_p \) and \( lastLinked_p \) and return it at line 401. Otherwise, \( p \) must access \( B \), update \( lastValue_p \) and \( lastLinked_p \) depending on the outcome, and compute its response accordingly (lines 404–411).

The invariant that relates the state of \( B \) and the state of the private variables \( lastValue \) and \( lastValue \) for a process is captured by the following lemma:

\textbf{Lemma 12.4.} For any history \( H \) of \( I_E^{BL} \) or \( I_E^{EW} \) where Condition 6.4 holds, if a process \( p \) applies a \( B\text{.hasChanged()} \) operation with response \textit{false} in step \( i \) of \( H \), then in states \( H[i - 1] \) and \( H[i] \) of \( H \) \( lastValue_p = B.V \) and \( lastLinked_p = B.Linked[p] \).

\textit{Proof.} First, note that if \( p \) applies an operation execution of a type other than \( \text{hasChanged()} \) on \( B \), say in step \( j \) of \( H \), then \( B.V = lastValue_p \) and \( B.Linked[p] = lastLinked_p \) in state \( H[j] \). This follows from the structure of our access procedures for \( I_E^{BL} \) and \( I_E^{EW} \) (recall our convention that accesses to local variables are atomic with the operation on \( B \) that precedes them in an access procedure): For \text{Read}, \( p \) calls \( B\text{.Read()} \) at line 385 and assigns the response to \( lastValue_p \). In this case, since \( p \)'s call to \( B\text{.hasChanged()} \) at line 384 returns \textit{true}, some process \( q \neq p \) applied a write-like operation on \( B \) after \( p \)'s last operation different from \( \text{hasChanged()} \) on \( B \), which implies that \( B.Linked[p] = \text{false} \) in state \( H[j] \). Accordingly, \( p \) assigns \( lastLinked_p = \text{false} \) at line 386. For \text{LL}, \( p \) calls \( B\text{.LL()} \) at line 394 and assigns the response to \( lastValue_p \). Then, \( p \) assigns \( lastLinked_p = \text{true} \) at line 395, which is correct in the case of a \text{LL}. For \text{Write(new)} (in the case of \( I_E^{BL} \)), \( p \) calls \( B\text{.Write(new)} \) at line 389 and assigns \( lastValue_p = \text{new} \) at line 390. Then, \( p \) assigns \( lastLinked_p = \text{false} \) at line 391, which is correct in the case of a \text{Write}. For \text{ECAS(isSC, cmp, new)}, \( p \) calls \( B\text{.ECAS(isSC, cmp, new)} \) at line 404 and then assigns \( lastValue_p = \text{new} \) at line 406 on success, otherwise assigns \( lastValue_p = \text{old} \) at line 408.
Declarations.

Shared variables: (global)

$B$ – ECAS object obtained using either $I_{BE}^P$ or $I_{BEW}^P$, initialized the same way as this implementation (with $Crumb[1..N] = \text{false}$ in the initial block)

Private variables: (per-process)

$lastValue$ – holds a value of the target object, uninitialized

$lastLinked$ – Boolean, uninitialized

$old$ – value of target object, uninitialized

$succ$ – Boolean, uninitialized

Function $\text{Read}()$

384 if $B$.hasChanged() = true then
385 | $lastValue := B$.Read()
386 | $lastLinked := \text{false}$
387 end
388 return $lastValue$

Function $\text{LL}()$

393 if $B$.hasChanged() = true $\lor$
394 | $lastLinked = \text{false}$ then
395 | $lastValue := B$.LL()
396 end
397 return $lastValue$

Function $\text{Write(new)}$

389 $B$.Write(new)
390 $lastValue := \text{new}$
391 $lastLinked := \text{false}$
392 return $\text{OK}$

Function $\text{ECAS(isSC, cmp, new)}$

398 if $B$.hasChanged() = false then
399 | $old := lastValue$
400 | if ($isSC = \text{true} \land lastLinked = \text{false}$) $\lor$
401 | ($isSC = \text{false} \land cmp \neq old$) then
402 | | return $\langle old, \text{false} \rangle$ // Operation execution failed.
403 | end
404 | $\langle old, succ \rangle := B$.ECAS($isSC, cmp, new$)
405 if $succ = \text{true}$ then
406 | $lastValue := \text{new}$
407 else
408 | $lastValue := old$
409 end
410 $lastLinked := \text{false}$
411 return $\langle old, succ \rangle$

Figure 12.2: Implementations $I_{BL}^{BE}$ and $I_{BL}^{BEW}$ for the write-through CC model.
where \textit{old} is the value of \textit{B} returns by \textit{p}'s \textit{B.ECAS}. Then, \textit{p} assigns \textit{lastLinked}_p = \text{false} at line 410, which is correct in the case of an \textit{ECAS} because Condition 6.4 holds in \textit{H}, and so either \textit{isSC} = \text{false} and \textit{p} never applied a \textit{B.LL()}, or \textit{isSC} = \text{true} and \textit{B.Linked}[p] = \text{false} just after \textit{p}'s \textit{B.ECAS} at line 404 whether that \textit{ECAS} succeeds or not.

Now consider \textit{p}'s call to \textit{B.hasChanged()} with response \text{false} in step \textit{i} of \textit{H}. Since the response is \text{false}, \textit{p} must have accessed \textit{B} in \textit{H} before step \textit{i} by applying an operation other than \textit{hasChanged()}. Suppose that this last occurred in step \textit{j} < \textit{i}. Then as we proved above, \textit{B.V} = \text{lastValue}_p and \textit{B.Linked}[p] = \text{lastLinked}_p in state \textit{H[j]}. The lemma then follows because by the response of \textit{p}'s call to \textit{B.hasChanged()} in step \textit{i}, neither \textit{B.V} nor \textit{B.Linked}[p] changes between steps \textit{j} and \textit{i}. The latter point follows because \textit{p}'s \textit{B.hasChanged()} in step \textit{i} returned \text{false}, and so all base object operations on \textit{B} strictly between steps \textit{j} and \textit{i} are read-like and belong to processes other than \textit{p}, hence they do not change \textit{B.V} or \textit{B.Linked}[p].

\textbf{Theorem 12.5.} The implementations \textit{I}^{BL}_E and \textit{I}^{BL}_EW satisfy Specifications 6.1 (linearizability) and 6.2 (termination) under Condition 6.4. Furthermore, each operation execution on the target object incurs \(O(1)\) RMRs in the write-through CC model.

\textit{Proof.} Specification 6.2 (termination) and \(O(1)\) RMR complexity follow from the structure of the access procedures, and the fact that the underlying implementation of \textit{B} (i.e., either \textit{I}^{B}_E+ or \textit{I}^{B}_EW+) requires \(O(1)\) RMRs per operation. Now consider Specification 6.1 (linearizability). For any finite history \textit{H} of \textit{I}^{BL}_E or \textit{I}^{BL}_EW where Condition 6.4 holds, we construct a candidate linearization \(\bar{H}\) as follows. First, for each operation execution \textit{Op} on the target object, assign an integer timestamp \textit{s} as follows:

\textbf{Definition 12.6.} The timestamp \textit{s} for an arbitrary operation execution \textit{Op} in \textit{H}, say by process \textit{p}, and its completion (where applicable), are defined as follows:

\begin{enumerate}
\item[(a)] If \textit{p} applies an operation other than \textit{hasChanged()} on \textit{B} during \textit{Op} in step \textit{i} of \textit{H}, then \textit{s} = \textit{i}.
\hspace{0.5cm} (If \textit{Op} is pending in \textit{H}, its completion is the response of this operation.)

\item[(b)] Else if \textit{Op} is complete and \textit{p} applies a \textit{hasChanged()} on \textit{B} during \textit{Op} in step \textit{i} of \textit{H}, then \textit{s} = \textit{i}.

\item[(c)] Otherwise \textit{s} is undefined.
\end{enumerate}
To form $\overline{H}$, we order operation executions for which timestamps are defined in ascending order of timestamp. Operation executions that are pending in $H$, and whose timestamps are defined, are completed as explained in Definition 12.6. It follows easily that $\overline{H}$ satisfies properties (a) and (b) of linearizability (sequential completion and order preservation), so consider property (a) (conformity to type $\tau_{ECAS}$). Since conformity to a type is a safety property it suffices to consider finite $\overline{H}$. Let $\nu_i$ denote the $i$'th operation execution in $\overline{H}$, $s_i$ its timestamp, and $k$ the length of $\overline{H}$. Define $s_0 = 0$ and $s_{k+1} = \infty$. Let $\nu_i$ for $0 \leq i \leq k$ denote the state of a correctly implemented ECAS object after applying the first $i$ operation executions from $\overline{H}$. Let $\nu_i.V$ and $\nu_i.Linked[1..N]$ denote the two components of the state represented by $\nu_i$. Similarly, let $B.V$ and $B.Linked[1..N]$ denote the two components of the state of base object $B$.

We will prove that for any $i \in \mathbb{N}$, $0 \leq i \leq k$:

(a) For any integer $s \in [s_i, s_{i+1})$, $B.V = \nu_i.V$ holds in state $H[s]$.

(b) For any integer $s \in [s_i, s_{i+1})$, $B.Linked[1..N] = \nu_i.Linked[1..N]$ holds in state $H[s]$.

(c) If $i > 0$, then the response of $Op_i$ is correct in $\overline{H}$.

Part (c) implies the lemma, but we require all three parts for induction. Now let $S(i)$ denote parts (a)–(c) for a particular value of $i$. Note that in $H$, the state of $B$ is changed only by an execution of lines 394, 389 or 404, which is an atomic step that defines the timestamp of an operation execution (on the target object) in $\overline{H}$. Therefore, the state of $B$ does not change between atomic steps $s_i$ and $s_{i+1}$ in $H$. This, in turn, implies that to prove parts (a)–(b) of $S(i)$, it suffices to prove that $B = \nu_i$ in state $H[s_i]$ – and that is all we do in the inductive proof that follows.

For $S(0)$, (a) and (b) follow from our earlier definition of $s_0$ and the initialization of $B$ to $\nu_0$ – the initial state for type $\tau_{ECAS}$. Part (c) holds trivially for $S(0)$. Now for any $i$, $0 < i \leq k$, suppose that $S(i - 1)$ holds, and consider $S(i)$. We proceed by cases on how $s_i$ was obtained. In our analysis we rely on the observation that if step $s_i$ in $H$ is a $B$.hasChanged() operation then its return value is false. (This follows from Definition 12.6 and the algorithm.)

**Case A:** Process $p$ calls $B$.hasChanged() at line 384 of Read with response false in step $s_i$ of $H$. Here $\nu_i = \nu_{i-1}$ and step $s_i$ does not change the state of $B$, so $S(i)$ (a) and $S(i)(b)$ follow from $S(i - 1)$. For $S(i)$ (c), note that $Op_i$ returns the value
read from \( \text{lastValue}_p \), which equals \( B.V \) in state \( H[s_i - 1] \) by Lemma 12.4 since \( p \)'s call to \( B.\text{hasChanged()} \) at line 384 returns \text{false}. Thus, \( Op_i \) returns \( \nu_{i-1}.V \) by \( S(i - 1) \) (a), as wanted.

**Case B:** Process \( p \) calls \( B.\text{Read()} \) at line 385 of \text{Read} in step \( s_i \) of \( H \). Here \( \nu_i = \nu_{i-1} \) and step \( s_i \) does not change the state of \( B \), so \( S(i) \) (a) and \( S(i)(b) \) follow from \( S(i - 1) \). For \( S(i) \) (c), note that \( Op_i \) returns the value of \( B.V \) in state \( H[s_i - 1] \) by the algorithm, which equals \( \nu_{i-1}.V \) by \( S(i - 1) \) (a), as wanted.

**Case C:** Process \( p \) calls \( B.\text{hasChanged()} \) at line 393 of \text{LL}, with response \text{false}, in step \( s_i \) of \( H \). Here the test at line 393 fails (otherwise \( s_i \) would correspond to the \( B.\text{LL} \) atomic step at line 394) and so \( \text{lastLinked}_p = \text{true} \) and \( B.\text{Linked}[p] = \text{true} \) in state \( H[s_i - 1] \) by Lemma 12.4. Consequently, \( \nu_{i-1}.\text{Linked}[p] = \text{true} \) by \( S(i - 1) \) (b). Thus, \( \nu_i = \nu_{i-1} \) holds, and we proceed as in Case A.

**Case D:** Process \( p \) calls \( B.\text{LL()} \) at line 394 of \text{LL} in step \( s_i \) of \( H \). The analysis is analogous to Case B.

**Case E:** Process \( p \) calls \( B.\text{Write(new)} \) at line 389 of \text{Write} in step \( s_i \) of \( H \), where \text{new} is the argument of \( Op_i \). Here \( \nu_i.V = \text{new} \) and \( \nu_i.\text{Linked}[1..N] = \text{false} \). \( S(i) \) (a) and \( S(i)(b) \) follow since \( B.V = \text{new} \) and \( B.\text{Linked}[1..N] = \text{false} \) by the action of step \( s_i \). \( S(i) \) (c) holds since \( Op_i \) is a \text{Write} and returns OK.

**Case F:** Process \( p \) calls \( B.\text{hasChanged()} \) at line 398 of \text{ECAS}(\text{isSC}, \text{cmp}, \text{new}) \) with response \text{false} in step \( s_i \) of \( H \). It follows that the test at line 400 succeeds during \( Op_i \), otherwise \( s_i \) would correspond to the \( B.\text{ECAS} \) atomic step at line 404. Thus, either \( \text{isSC} = \text{true} \) and \( \text{lastLinked}_p = \text{false} \), or \( \text{isSC} = \text{false} \) and \( \text{cmp} \neq \text{old} \).

**Subcase F-i:** \( \text{isSC} = \text{true} \) and \( \text{lastLinked}_p = \text{false} \) holds. Then \( B.\text{Linked}[p] = \text{false} \) holds in state \( H[s_i - 1] \) by Lemma 12.4, and so \( \nu_{i-1}.\text{Linked}[p] = \text{false} \) holds by \( S(i - 1) \) (b). Since \( \text{isSC} = \text{true} \), \( Op_i \) fails and \( \nu_i = \nu_{i-1} \) holds. Since step \( s_i \) does not change the state of \( B \), \( S(i) \) (a) and \( S(i) \) (b) both follow from \( S(i - 1) \). For \( S(i) \) (c) note that the correct response for \( Op_i \) is \( \langle \nu_{i-1}.V, \text{false} \rangle \). \( Op_i \) returns exactly this at line 401 since here \( \text{old} \) is the value of \( \text{lastValue}_p \) in state \( H[s_i - 1] \), which equals \( B.V \) in state \( H[s_i - 1] \) by Lemma 12.4 and hence equals \( \nu_{i-1}.V \) by \( S(i - 1) \) (a).

**Subcase F-ii:** \( \text{isSC} = \text{false} \) and \( \text{cmp} \neq \text{old} \). Since \( \text{old} = \nu_{i-1}.V \) as in Subcase F-i, \( \text{cmp} \neq \text{old} \) implies that \( Op_i \) fails and so \( \nu_i = \nu_{i-1} \) holds. Thus, \( S(i) \) (a) and \( S(i) \) (b)
both follow from \( S(i - 1) \). \( S(i) \) (c) follows as in Subcase F-i.

**Case G:** Process \( p \) calls \( B.ECAS(isSC, cmp, new) \) at line 404 of \( ECAS(isSC, cmp, new) \) in step \( s_i \) of \( H \). Here \( B = \nu_{i-1} \) in state \( H[s_i - 1] \) by \( S(i - 1) \) (a)–(b), and so \( B = \nu_i \) in state \( H[s_i] \) by the action of step \( s_i \). Thus, \( S(i) \) (a) and \( S(i) \) (b) both hold. For \( S(i) \) (c) note that the correct response of \( Op_i \) is \( \langle \nu_{i-1}.V, true \rangle \) if \( Op_i \) succeeds and \( \langle \nu_{i-1}.V, false \rangle \) if \( Op_i \) fails. Since \( B = \nu_{i-1} \) in state \( H[s_i - 1] \), this equals the response of \( p \)'s call to \( B.ECAS \) in step \( s_i \), and hence the response of \( Op_i \) at line 411.

\[ \square \]

**Theorem 12.7.** The implementations \( I_{E}^{BL} \) and \( I_{EW}^{BL} \) satisfy the locality property in the write-through CC model (see Definitions 7.2).

**Proof.** Consider a history \( H \) of \( I_{E}^{BL} \) or \( I_{EW}^{BL} \). Let \( \bar{H} \) be the linearization of \( H \) as defined in our proof of linearizability (Theorem 12.5). Fix process \( p \) and a sequence \( \bar{H}' \) of consecutive read-like operation executions in \( \bar{H} \). Let \( H' \) denote the sequence of base object atomic steps in \( H \) corresponding to \( \bar{H}' \).

It follows that no write-like atomic step on \( B \) occurs in \( H \) between the first and last steps (inclusive) of \( H' \), otherwise if this occurred in step \( i \), then there would be a write-like operation execution in \( \bar{H}' \) with timestamp \( i \). Consequently, by the structure of the access procedures, all of \( p \)'s atomic steps on \( B \) in \( H' \) except possibly the first are \( B.\text{hasChanged()} \) atomic steps that return \( false \). By Theorem 12.3, \( p \) incurs \( O(1) \) RMRs performing these atomic steps, and so \( p \) incurs \( O(1) \) RMRs in \( H \) applying all of its steps from \( H' \). \[ \square \]
12.3 DSM Model

Applying our memory management techniques described earlier in Chapter 11 to a locally-accessible ECAS implementation in the DSM model breaks the locality property. This is because the designated process $p_{\text{special}}$ (to which the ECAS implementation is local) accesses remote variables in several places while executing the functions introduced in Sections 11.1 and 11.3, in particular AfterOp() (see Figure 11.3), MiddleOp$(x,y)$ (see Figure 11.4) and RecycleBlock() (see Figure 11.7). In AfterOp(), a process must on occasion write a remote variable at line 328 in order to signal another process that may be waiting for it at line 343 of MiddleOp$(x,y)$. The same occurs at line 333 of MiddleOp$(x,y)$, and line 357 of MiddleOp-ExitCS$(y)$ (which is called at line 349 of MiddleOp$(x,y)$). In RecycleBlock(), a process may access some register in a block remotely at lines 364, 365, 373, 375 and 376. (There are also several places where processes access shared variables whose locality has not been specified; these variables can be made local to $p_{\text{special}}$ but this would not suffice for our purposes in this section.)

Applying our memory management techniques to our locally-accessible ECAS implementations in the DSM model poses two technical challenges: (1) In MiddleOp$(x,y)$, we cannot use the “waiting mechanism” at lines 336–346 when process $p_{\text{special}}$ is the one being waited for (because $p_{\text{special}}$ cannot signal the waiting process without performing an RMR), and yet we need some mechanism to ensure that when a block is being reset by a non-special process it is not accessed concurrently by $p_{\text{special}}$ (at least not in a way that interferes with the reset). (2) In RecycleBlock(), $p_{\text{special}}$ cannot reset arbitrary registers in a block (because some registers are remote to $p_{\text{special}}$), and yet we need $p_{\text{special}}$ to be able to reset blocks (e.g., in a history where $p_{\text{special}}$ is the only process that allocates blocks). We overcome the first challenge by introducing a new synchronization mechanism between $p_{\text{special}}$ and the non-special process that is resetting a block inside RecycleBlock(). We overcome the second challenge by having non-special processes help $p_{\text{special}}$ reset blocks.

To simplify exposition, we will describe separate block allocators for the two categories of processes under consideration in this section: $p_{\text{special}}$, and non-special processes. By “separate” we mean that in each case there is a separate implementation of the function AllocBlock() and one or more recycling functions (see Definition 11.6). We refer to these as the special allocator and non-special allocator, and we refer to blocks returned by them as special blocks and non-special blocks. Note that $p_{\text{special}}$ may access non-special
blocks and similarly non-special processes may access special blocks, which is necessary in light of the way blocks are used: the process that allocates and makes current a block is not necessarily the same process that accesses the block later on. Thus, in our analysis of the allocator for one category of processes, we will still refer to processes of the other category in order to establish the correctness properties.

Blocks from the two allocators are managed in very different ways. The non-special allocator is very similar to the one described in Chapter 11 and so blocks obtained from it are accessed and reset as described in Section 11.2 (i.e., by tracking block and register versions, and by using a combination of soft and hard resets). In contrast, the special allocator cannot follow closely the approach described in Chapter 11 because of the two technical challenges mentioned earlier. Instead, in this allocator multiple processes reset (parts of) a block in parallel, and each process is responsible for resetting the registers it has accessed in a block when that block is reset. Thus, whereas non-special processes are responsible for recycling non-special blocks, special blocks are reset jointly by all processes. Another difference between the special and non-special allocators is that soft resets are not needed for special blocks; hard resets alone are sufficient because they can be carried out at a cost of $O(1)$ RMRs per process. (This is because blocks are reset by many processes in parallel and because only those registers that have been accessed in a block are reset.) Thus, we can dispense with soft resets entirely, as well as with the associated mechanism for accessing blocks that uses block versions and register versions. (This simpler approach cannot be used in the non-special allocator because it relies on the ability of one process to wait for every other process to finish resetting a block, as shown in Figure 12.3, which is costly in terms of RMRs when the waiting process is non-special.)

In the special case of the writable ECAS implementation, when a block obtained from $p_{\text{special}}$'s allocator is reset, the base object $B$ is treated differently from other shared objects in the block. That is, process $p_{\text{special}}$ always resets $B$, and no other process accesses $B$ in parallel. (I.e., the burden of resetting $B$ is shifted entirely from non-special processes to $p_{\text{special}}$.) Process $p_{\text{special}}$ resets $B$ using the special operation type \texttt{Reset() } discussed at the end of Section 11.2, which is why we must ensure other processes do not concurrently access $B$: the access procedure proposed for \texttt{Reset()} is correct only if $B$ is accessed in mutual exclusion. A subtle point that deserves more attention here is that $p_{\text{special}}$ must be able to apply \texttt{Reset()} without incurring any RMRs. Although we assume in the case of the writable ECAS implementation that $B$ is local to $p_{\text{special}}$, our
analysis of locality in Chapter 7.3 did not take into account operation type \(\text{Reset}()\). For completeness, we point out that since the access procedure for \(\text{Reset}()\) is very similar to the access procedure for \(\text{Write}(\text{new})\), calls to \(\text{Reset}()\) by \(p_{\text{special}}\) incur zero RMRs by the same arguments as used for analyzing \(\text{Write}(\text{new})\).

Since any process may access blocks from either of the two allocators under consideration, and since blocks from the two allocators are accessed in very different ways (i.e., in one case using block versions and register versions, in the other case directly), another subtle issue is that a process must be able to determine whether a particular block is special or non-special. For this purpose we introduce a block field called \(\text{isSpecial}\) that stores a Boolean indicating whether the block is special. This register is accessed directly (i.e., without using block and register versions) and it is not touched during a reset because its value is constant. We assume that the initial block is non-special.

### 12.3.1 Non-special Allocator

The non-special allocator consists of the functions \(\text{AllocBlock}()\) and \(\text{RecycleBlock}()\), which are referred to by Specifications 11.7–11.8, as well as the functions \(\text{BeforeOp()}\), \(\text{AfterOp()}\) and \(\text{MiddleOp}(x, y)\), which were introduced in Section 11.1. These functions are called only by non-special processes, and they have no counterparts that might be called by \(p_{\text{special}}\). (Note, however, that both \(p_{\text{special}}\) and non-special processes do call such functions in the special allocator, described in Section 12.3.2.) This breaks the assumptions underlying our analysis in Sections 11.1 and 11.3 because \(p_{\text{special}}\) may access non-special blocks without calling \(\text{BeforeOp()}\) and \(\text{AfterOp()}\), and because \(p_{\text{special}}\) may perform successful \(\text{chngCurBlock}(x, y)\) operations without calling the subroutine \(\text{MiddleOp}(x, y)\) the way a non-special process would (i.e., immediately after a successful \(\text{chngCurBlock}(x, y)\)). Consequently, the algorithms for non-special processes must be modified accordingly. We also make some changes to the execution path for \(p_{\text{special}}\), as explained shortly.

The problem with \(p_{\text{special}}\) calling \(\text{chngCurBlock}(x, y)\) but not \(\text{MiddleOp}(x, y)\) is that the proof of Theorem 11.20 breaks. Specifically, a non-special process \(q\) may loop forever at line 352 of \(\text{MiddleOp}-\text{EnterCS}(x)\) (which is called at line 335 of \(\text{MiddleOp}(x, y)\)), waiting for \(p_{\text{special}}\) to write the Boolean flag \(x \triangleright \text{CSWaitFlag}[q]\) after \(p_{\text{special}}\) allocated \(x\) and made it current. To prevent this, it suffices to make a small change to the protocol for calling \(\text{MiddleOp}(x, y)\). That is, instead of calling \(\text{MiddleOp}(x, y)\) immediately
Chapter 12. Space Complexity in Locally-Accessible ECAS

206

after a call to `chngCurBlock(x, y)`, a non-special process instead calls `MiddleOp(x', y)`, where `x'` is the last non-special block that became current before `x` (possibly the initial block). Determining the correct block address `x'` is straightforward if we introduce a new block field, say `lastNonSpecialBlock`, and update it as follows: For the initial block `b_0`, `lastNonSpecialBlock = b_0`. Then, a process calling `chngCurBlock(x, y)` assigns `y ⊲ lastNonSpecialBlock` just before calling `chngCurBlock(x, y)`. For non-special process, the value assigned is simply `y`. For `p_special`, the value is simply copied over from `x ⊲ lastNonSpecialBlock`. Finally, if the call to `chngCurBlock(x, y)` succeeds, then in the subsequent call to `MiddleOp(x', y)`, the argument `x'` is taken to be `x ⊲ lastNonSpecialBlock`.

The problem with `p_special` accessing non-special blocks without calling `BeforeOp()` and `AfterOp()` is that the proof of Lemma 11.28 breaks. Specifically, a block `x` to which `p_special` holds a (strong) reference may be added to the list `DirtyBlocks` of blocks maintained internally by the allocator, despite `ref(x)` being positive (see Figures 11.6–11.7). To remedy this, non-special processes require a new mechanism (i.e., other than the “waiting mechanism” at lines 338–346 of `MiddleOp`) to ensure that `ref(x) = 0` before they attempt to recycle block `x`. Specifically, when the head element `x` of `RecentBlocks` is considered for recycling at line 362 of `RecycleBlock()`, it suffices for the caller of `RecycleBlock()` to check whether `p_special` holds a reference to `x`, and if so, choose an alternate block for recycling. The alternate block could be the next element after `x` in `RecentBlocks`, provided we increase the number of blocks initially in `RecentBlocks` by one. This small change ensures that the first and second block in `RecentBlocks` is recyclable when a process reaches line 362 of `RecycleBlock()`. Unfortunately, it is difficult to determine which block `p_special` is accessing because `p_special` cannot atomically execute `getCurBlock()` and record the response of this call in a shared object that can be read by a non-special process `q` inside `RecycleBlock()`. Instead, we devise a special shared object for synchronization between `q` and `p_special` that allows `q` and `p_special` to agree on which block `p_special` will access in a given operation execution on the ECAS object.

The new shared object used by `p_special` and `q` is accessed in a manner that satisfies Condition 10.2 (i.e., by at most one non-special process at a time), and so it can be constructed using the universal method from Chapter 10. (This is because the object is accessed by non-special processes only in `RecycleBlock()`, which is executed in mutual exclusion by Condition 11.10 (a).) The type of this object is \( \tau_{CAS} \), as defined in Chapters 6, and supports operation type \( \text{CAS}(cmp, new) \). It is used to store either a block
address or one of two special values: $\bot$ and $\top$. The initial value is $\bot$, which indicates that \( p_{\text{special}} \) is not accessing any block. A value of $\top$ indicates that \( p_{\text{special}} \) is about to access some block, but has not yet announced which one, possibly because it does not know yet (i.e., \( p_{\text{special}} \) has not yet applied a \texttt{getCurBlock()} on the block manager). A value equal to a block address \( b \) indicates that \( p_{\text{special}} \) is accessing block \( b \) during its operation execution on the ECAS object.

The shared CAS object, subsequently denoted \( C \), is accessed as follows. When \( p_{\text{special}} \) begins executing an access procedure of the ECAS implementation, it applies a \texttt{CAS(\bot, \top)} (which always succeeds), indicating its intent to access some block. After completing a call to \texttt{getCurBlock()}, say with response \( x \), \( p_{\text{special}} \) applies a \texttt{CAS(\top, x)}. If the CAS succeeds, then no non-special process is trying to reset \( x \) inside \texttt{RecycleBlock()}, and so \( p_{\text{special}} \) is free to access \( x \). If the CAS fails, then block \( x \) ceased to be current since \( p_{\text{special}} \)'s \texttt{getCurBlock()} and moreover it has undergone recycling. Consequently, \( p_{\text{special}} \) must not access \( x \). Instead, \( p_{\text{special}} \) adopts the response of its failed CAS as the address of the block it should access. (This value is set by a non-special process, as explained shortly.) Now let \( y \) be the block \( p_{\text{special}} \) has chosen to access. At the end of its access procedure, \( p_{\text{special}} \) applies a \texttt{CAS(y, \bot)} to \( C \) (which always succeeds), indicating that it is no longer accessing any block.

Non-special processes access \( C \) only inside \texttt{RecycleBlock()} (see Figure 11.7). Before dequeuing a block from \texttt{RecentBlocks} at line 362 of \texttt{RecycleBlock()}, a non-special process \( q \) identifies the head element \( b \) of \texttt{RecentBlocks} and its successor \( b' \). The successor is needed in case \( p_{\text{special}} \) is accessing \( b \), in which case \( q \) cannot process \( b \) as before (i.e., attempt to soft-reset \( b \)). Process \( q \) then calls \texttt{getCurBlock()}, say with response \( y \), and applies \texttt{CAS(\top, y)} to \( C \). If \( q \)'s CAS fails with a response that is either \( \bot \) or the address of a block different from \( b \), then \( q \) is free to dequeue \( b \) from \texttt{RecentBlocks} and proceed as before from line 362 of \texttt{RecycleBlock()}. This is because \( p_{\text{special}} \) is either not applying an operation execution on the ECAS object, or is applying one where it accesses some block different from \( b \). Similarly, if \( q \)'s CAS succeeds then \( q \) will not access block \( b \) (because it will access \( y \) instead as explained earlier), and so \( q \) proceeds as before. On the other hand, if \( q \)'s CAS fails with response \( b \) then \( q \) cannot proceed as before as that would create the possibility that \( q \) soft-resets \( b \) (or that some non-special process hard-resets \( b \) later on) while \( p_{\text{special}} \) is accessing \( b \). Instead, \( q \) removes \( b' \) (i.e., \( b \)'s successor) from \texttt{RecentBlocks}, and continues executing after line 362 as if it has dequeued \( b' \) there from the head of \texttt{RecentBlocks}. It is safe for \( q \) to do so because \( b' \) is recyclable just like \( b \), as
explained earlier.

Because in some cases \( q \) tells \( p_{\text{special}} \) which block it should access (rather than \( p_{\text{special}} \) discovering this directly from the block manager), our analysis in the proof of linearizability of the ECAS implementation must change slightly. That is, to assign a timestamp to an operation execution \( Op \) by \( p_{\text{special}} \) where it calls \( \text{getCurBlock()} \) with response \( x \) and then applies a failed \( \text{CAS}(\top, x) \) with response \( y \), we treat \( y \) as the response of \( \text{getCurBlock()} \) rather than \( x \). For example, in Definition 6.8 (a) the timestamp would be \( (y, i, 0) \) instead of \( (x, i, 0) \), where \( i \) is the index of the step where \( p_{\text{special}} \) reads \( y \uparrow V \).

A subtle but important issue in the non-special allocator is what happens to Conditions 11.9–11.11. We do rely on these for correctness of the non-special allocator, but they must be modified slightly because \( p_{\text{special}} \) is “out of the picture” in this context. Specifically, we ignore calls to \( \text{AllocBlock()} \) and \( \text{RecycleBlock()} \) by \( p_{\text{special}} \) in Condition 11.9 (b) and Condition 11.10 (b)–(c) because \( p_{\text{special}} \) does not invoke the non-special allocator. Similarly, in Conditions 11.10 (c) and 11.11 (b) we consider the \( i \)’th successful \( \text{chgCurBlock}(x, y) \) by a non-special process, and the \( i \)’th non-special block made current.

We now give a brief analysis of the non-special allocator. First, consider the instrumentation of the ECAS implementation, which is as in Section 11.1 except that \( p_{\text{special}} \) does not call \( \text{BeforeOp()} \), \( \text{AfterOp()} \) or \( \text{MiddleOp}(x, y) \), and function \( \text{MiddleOp}(x, y) \) for non-special processes has changed slightly, as described earlier. The analogs of Theorems 11.17–11.19 follow as in Section 11.1, except that in Theorem 11.18 and 11.19 we consider slightly modified versions of Conditions 11.10 (c) and 11.11 (b), as described earlier. The analog of Theorem 11.20 follows as in Section 11.1 except that in the analysis of the busy-wait loop in \( \text{MiddleOp-EnterCS}(x) \) (i.e., Case B) we take into account the change in how the arguments of \( \text{MiddleOp}(x, y) \) are chosen (\( x \) is the last non-special block made current before \( y \)). The analogs of Theorems 11.21 and 11.22 follow as in Section 11.1.

Next, consider the correctness properties of the non-special block allocator. As in Section 11.3 we model executions of functions \( \text{AllocBlock()} \) and \( \text{RecycleBlock()} \) using a concurrent system \( S \), and we show that histories of \( S \) satisfy Specifications 11.7 and 11.8 provided that Conditions 11.9–11.11 (amended as described above) hold. We also show that each call to \( \text{AllocBlock()} \) or \( \text{RecycleBlock()} \) incurs \( O(1) \) RMRs.

**Theorem 12.8.** For any history of \( S \) where Conditions 11.9–11.11 (amended as de-
scribed above) hold, Specifications 11.7–11.8 also hold. Furthermore, every call by a non-special process to AllocBlock() or to RecycleBlock() incurs $O(1)$ RMRs in the DSM model.

Proof. Specification 11.7 follows by the same sequence of lemmas as in Section 11.1, with some changes: For the analogs of Lemmas 11.23 and 11.24, we consider one additional block. For the analog of Lemma 11.28, we must deal with the special case where a process removes the second element of RecentBlocks in RecycleBlock() instead of a head element because it has discovered (using the shared object $C$) that $p_{\text{special}}$ may hold a strong reference to the block at the head of RecentBlocks. In particular, we must argue that the reference count for the second element of RecentBlocks is zero, using the fact that RecentBlocks initially contains one more element than before.

Specification 11.8 follows as in the proof of Theorem 11.32 because histories of $S$ contain calls to AllocBlock() and RecycleBlock() only by non-special processes. Similarly, $O(1)$ RMR complexity of AllocBlock() and RecycleBlock() follows from the structure of these functions, which are the same as in Section 11.3 except that a non-special process performs $O(1)$ additional RMRs in each call to RecycleBlock() accessing the shared object $C$, the queue RecentBlocks, and the block manager. □

Theorem 12.9. The non-special block allocator uses $3N + 6$ blocks.

Proof. The non-special allocator uses one more block than the allocator presented in Section 11.3, which is $3N + 5$ by Theorem 11.33. □

12.3.2 Special Allocator

The special allocator consists of AllocBlock() for process $p_{\text{special}}$, as well as two recycling functions: RecycleBlockS(), which $p_{\text{special}}$ calls after each call to AllocBlock(), and RecycleBlockNS(), which non-special processes call before each operation execution on the target object. (Note that we no longer distinguish between visible and invisible calls to AllocBlock() by $p_{\text{special}}$.) The recycling functions must be called according to the following etiquette:

Condition 12.10. For any history $H$, calls to AllocBlock(), RecycleBlockS() and RecycleBlockNS() satisfy the following:
(a) only process $p_{\text{special}}$ calls $\text{RecycleBlockS()}$, and only non-special processes call $\text{RecycleBlockNS()}$; and

(b) process $p_{\text{special}}$ calls $\text{AllocBlock()}$ and $\text{RecycleBlockS()}$ in an alternating sequence, beginning with $\text{AllocBlock()}$; and

(c) each non-special process calls $\text{RecycleBlockNS()}$ at the beginning of every operation execution.

To support the special allocator we instrument the ECAS implementation under consideration by introducing functions similar in spirit to those described in Section 11.1. We describe details of the instrumentation and the block allocator separately in the next two sections.

### Instrumentation

In this section we describe how to instrument an ECAS implementation to support the special allocator. We assume that the implementation is already instrumented to support the non-special allocator, which means that non-special processes call $\text{BeforeOp()}$, $\text{AfterOp()}$ and $\text{MiddleOp}(x, y)$ as described in Sections 11.1 and 12.3.1. We now introduce three additional functions: $\text{BeforeOpNS()}$, $\text{AfterOpNS()}$ and $\text{MiddleOpS()}$. These are similar in spirit to $\text{BeforeOp()}$, $\text{AfterOp()}$ and $\text{MiddleOp}(x, y)$ but there are also some important differences. First, only non-special processes call $\text{BeforeOpNS()}$ and $\text{AfterOpNS()}$, namely before and after every operation execution on the target object. Second, $\text{MiddleOpS()}$ does not take any arguments, and is called only by $p_{\text{special}}$, namely after every call to $\text{chgCurBlock}(x, y)$ (successful or not) on the block manager.

The pseudo-code for $\text{BeforeOpNS()}$, $\text{AfterOpNS()}$ and $\text{MiddleOpS()}$ is presented in Figure 12.3. Inside $\text{BeforeOpNS()}$ and $\text{AfterOpNS()}$, non-special processes record state information using a pair of shared arrays: $\text{ActiveSpecial}[1..N]$ and $\text{WaitFlagSpecial}[1..N]$. These are analogous to $\text{Active}[1..N]$ and $\text{WaitFlag}[1..N]$ from Section 11.1. Process $p_{\text{special}}$ uses the information recorded in these arrays in $\text{MiddleOpS()}$ to deduce when the reference count for a block has reached zero. We adopt the convention that a process does not hold a reference to any block while executing $\text{AfterOpNS()}$, as we had done for $\text{AfterOp()}$ in Section 11.1.

We now begin our analysis. As regards the functions presented in Figure 12.3, our analysis consists (as in Section 11.1) of showing that the instrumented ECAS implementation satisfies certain useful properties in addition to linearizability, termination as well
Declarations

Shared variables: (global)
- $ActiveSpecial[1..N]$ – array of Boolean, all elements initially false and local to $p_{special}$
- $WaitFlagSpecial[1..N]$ – array of Boolean, all elements initially false and local to $p_{special}$

Private variables: (per-process)
- $q$ – process ID or ⊥, uninitialized

Function BeforeOpNS() for non-special processes
- RecycleBlockNS()
- write $ActiveSpecial[PID] := true$

Function AfterOpNS() for non-special processes
- write $ActiveSpecial[PID] := false$
- write $WaitFlagSpecial[PID] := true$

Function MiddleOpS() for process $p_{special}$

// Ensure that no non-special process holds a strong reference to a block that became current earlier than the last block made current by $p_{special}$.
- for each non-special process ID $q$ do
  - write $WaitFlagSpecial[q] := false$
  - if read($ActiveSpecial[q]$) = true then
    - await $WaitFlagSpecial[q] = true$
  - end
- end
- RecycleBlockS()

Figure 12.3: Subroutines BeforeOpNS(), AfterOpNS() and MiddleOpS().
as $O(1)$ RMR complexity and locality in the DSM model. The useful properties are Condition 12.10 (stated earlier) and the following two conditions:

**Condition 12.11.** For any history $H$, accesses to special blocks satisfy the following: for each special block $x$, if $\text{ref}(x)$ is continuously positive in some subhistory $H'$ of $H$, then in $H'$ $p_{\text{special}}$ accesses zero remote registers in $x$, and a non-special process accesses $O(1)$ remote registers in $x$ (ignoring any access to the base object $B$ in the case of a writable ECAS implementation).

**Condition 12.12.** For any history $H$, reference counts for special blocks satisfy the following:

(a) a non-special process holds a strong reference to at most one block in one operation execution on the ECAS object; and

(b) if the $i$’th block allocated by $p_{\text{special}}$ is $x$ then $\text{ref}(x)$ reaches zero after $p_{\text{special}}$ allocates $x$ and before $p_{\text{special}}$’s $(i + 2)$’nd call to $\text{RecycleBlockS}()$.

Regarding Conditions 11.9–11.11, we assume as in Section 12.3.1 that slightly modified versions of these hold. The special allocator depends on Conditions 11.9 (a) and 11.11 (a); the other conditions are needed only for the non-special allocator.

Let $I'_{E-DSM}$ and $I'_{EW-DSM}$ denote our locally-accessible ECAS implementations for the DSM model from Sections 7.3 and 8.2.2, respectively. Let $I'_{E-DSM}$ and $I'_{EW-DSM}$ denote the instrumented implementations obtained by transforming $I_{E-DSM}$ and $I_{EW-DSM}$, respectively, as described in Sections 11.1 and 12.3.1 to support the non-special allocator, and then transformed further by introducing $\text{BeforeOpNS}()$, $\text{AfterOpNS}()$ and $\text{MiddleOpS}()$ as described above. As in Sections 11.1, we assume that in $I'_{E-DSM}$ and $I'_{EW-DSM}$ the block allocator satisfies Specifications 11.7–11.8. We now establish Conditions 12.10–12.12 for $I'_{E-DSM}$ and $I'_{EW-DSM}$. (Conditions 11.9 (a) and 11.11 (a) hold by the same proofs as in Chapter 11 – see Theorems 11.17 and 11.19.)

**Theorem 12.13.** For any history $H$ of $I'_{E-DSM}$ or $I'_{EW-DSM}$, Condition 12.10 holds.

**Proof.**

**Part (a):** This follows because $\text{RecycleBlockS}()$ is called only at line 422 of $\text{MiddleOpS}()$, which only $p_{\text{special}}$ calls, and because $\text{RecycleBlockNS}()$ is called only at line 412 of $\text{BeforeOpNS}()$, which only non-special processes call.
**Part (b):** This follows because \texttt{RecycleBlockS()} is called only at line 422 of \texttt{MiddleOpS()}, and because \( p_{\text{special}} \) calls \texttt{AllocBlock()} and \texttt{MiddleOpS()} in an alternating sequence beginning with \texttt{AllocBlock()).}

**Part (c):** This follows because a non-special process calls \texttt{RecycleBlockNS()} at line 412 of \texttt{BeforeOpNS()}, and calls \texttt{BeforeOpNS()} at the beginning of every operation execution. \( \square \)

**Theorem 12.14.** For any history \( H \) of \( I_{E-DSM}' \) or \( I_{EW-DSM}' \), Condition 12.11 holds.

**Proof.** Let \( x \) be a special block. First, consider accesses by \( p_{\text{special}} \) to \( x \). If \( p_{\text{special}} \) accesses a remote register in \( x \) while \( \text{ref}(x) > 0 \), then this occurs outside of calls to \texttt{AllocBlock()} and any recycling function (in this case \texttt{RecycleBlockS()}) by Specification 11.7 (b). In that case, there is some history of the implementation \( I_{E-DSM} \) or \( I_{EW-DSM} \) (on which \( I_{E-DSM}' \) and \( I_{EW-DSM}' \), respectively, are based) where \( p_{\text{special}} \) accesses the same remote register, which contradicts the locality of \( I_{E-DSM} \) and \( I_{EW-DSM} \).

Next, consider accesses by a non-special process \( p \) to \( x \). Let \( H' \) be the subhistory of \( H \) where \( \text{ref}(x) > 0 \) holds continuously. As for \( p_{\text{special}} \), any access by \( p \) to \( x \) in \( H' \) occurs outside of calls to \texttt{AllocBlock()} and any recycling function (in this case \texttt{RecycleBlockNS()}) by Specification 11.7 (b). We must show that \( p \) accesses \( O(1) \) remote registers in \( x \) in \( H' \) despite the fact that there is a super-constant number of registers in \( x \) that are remote to \( p \) and that \( p \) may access in various histories (e.g., registers used in name consensus). It suffices to consider operations on the block manager because outside of these, \( p \) accesses \( O(1) \) registers in block \( x \) (i.e., \( x \triangleright writer \), \( x \triangleright V \) and \( x \triangleright \text{NextVal}[p] \) – see Figures 6.5–6.6 and Figure 8.2). There are \( O(1) \) registers that \( p \) accesses outside of the pseudo-lock (i.e., \( x \triangleright \text{winner} \), \( x \triangleright \text{helping} \), \( x \triangleright \text{specialDone} \), \( x \triangleright \text{helperDone} \) and \( x \triangleright A \) – see Figure 7.2).

Finally, consider the pseudo-lock. Since \( \text{ref}(x) > 0 \) holds continuously in \( H' \), it follows from Condition 4.1 that \( p \) calls each of \( x \triangleright \text{Pseudo-Enter}() \) and \( x \triangleright \text{Pseudo-Exit}() \) at most once in \( H' \), and so by the \( O(1) \) RMR complexity of the pseudo-lock in the DSM model (see Theorem 4.6) \( p \) accesses \( O(1) \) remote registers during each call. (In this context, we interpret Condition 4.1 as restricting the number of times a process calls \( x \triangleright \text{Pseudo-Enter}() \) and \( x \triangleright \text{Pseudo-Exit}() \) only between \( x \) being made current and being recycled, as explained in the proof of Theorem 11.12.) \( \square \)

**Theorem 12.15.** For any history \( H \) of \( I_{E-DSM}' \) or \( I_{EW-DSM}' \), Condition 12.12 holds.

**Proof.**
Part (a): A non-special process holds a strong reference to at most one block at a time because it calls \texttt{getCurBlock()} at most once in each operation execution.

Part (b): Suppose for contradiction that \( x \) is the \( i \)'th (special) block allocated by \( p_{\text{special}} \) and that \( \text{ref}(x) > 0 \) holds continuously from the state just after \( p_{\text{special}} \)'s \((i + 2)\)'nd call to \texttt{RecycleBlockS}(). Let \( H' \) be the corresponding subhistory of \( H \). Note that \( p_{\text{special}} \) calls \texttt{chgCurBlock(\ldots, \ldots)} in \( H' \) exactly twice, namely after its \((i + 1)\)'st and \((i + 2)\)'nd calls to \texttt{AllocBlock}(), which occur before \( p_{\text{special}} \)'s \((i + 2)\)'nd call to \texttt{RecycleBlockS}(). Consequently, a successful \texttt{chgCurBlock(\ldots, \ldots)} occurs in \( H' \) – either \( p_{\text{special}} \)'s call or call by a non-special process that causes \( p_{\text{special}} \)'s call to fail. Let \( H'' \) be the suffix of \( H' \) after this successful \texttt{chgCurBlock(\ldots, \ldots)}. Note that the analogs of Lemmas 11.13 and 11.14 hold for \( I'_{E-DSM} \) and \( I'_{EW-DSM} \) (by the same proofs as in Section 11.1). Consequently, \( x \) is not current at the beginning of \( H'' \) by the analog of Lemma 11.13. Furthermore, \( x \) does not become current again in \( H'' \) by the analog of Lemma 11.14 because \( \text{ref}(x) > 0 \) holds continuously in \( H'' \). Similarly, by Specification 11.7 (a) no process acquires a weak reference to \( x \) in \( H'' \) because \( \text{ref}(x) > 0 \) holds. Thus, our original supposition implies that some process \( q \) continuously holds a strong reference to \( x \) throughout \( H''. \) Process \( p_{\text{special}} \) does not do so because in \( H'' \) \( p_{\text{special}} \) completes the operation execution where it calls \texttt{AllocBlock()} for the \((i + 1)\)'st time, and \( q \) is non-special. Now consider \( p_{\text{special}} \)'s execution of lines 416–421 of \texttt{MiddleOpS()} in \( H' \). Let \( H''' \) denote this subhistory. Since we assume that \( q \) holds a strong reference to \( x \) continuously in \( H'' \), it does so in \( H''' \) as well, and furthermore in \( H''' \) \( q \) has completed a call to \texttt{BeforeOpNS()} but not yet made its subsequent call to \texttt{AfterOpNS()}. (Recall that a non-special does not acquire a reference to any block until after it completes \texttt{BeforeOpNS()} at the beginning of an operation execution, and that it holds no reference at all during a call to \texttt{AfterOpNS()} by our convention.) This implies that \( p_{\text{special}} \) reads \( \text{ActiveSpecial}[q] = \text{true} \) at line 418 because \( q \) has assigned \( \text{ActiveSpecial}[q] = \text{true} \) at line 413 of \texttt{BeforeOpNS()} and not yet assigned \( \text{ActiveSpecial}[q] = \text{false} \) at line 414 of \texttt{AfterOpNS()}. Next, \( p_{\text{special}} \) completes the busy-wait loop at line 419, after assigning \( \text{WaitFlagSpecial}[p] = \text{false} \) at line 417. This implies that some process assigns \( \text{WaitFlagSpecial}[p] = \text{true} \) in \( H''' \). By the algorithm, this can only happen if \( q \) executes line 415 of \texttt{AfterOpNS()}, which contradicts our earlier observation that \( q \) does not call \texttt{AfterOpNS()} in \( H''' \).

**Theorem 12.16.** For any fair history \( H \) of \( I'_{E-DSM} \) or \( I'_{EW-DSM} \), each call to \texttt{BeforeOpNS()},
AfterOpNS() and MiddleOpS() terminates.

Proof. For BeforeOpNS() and AfterOpNS(), termination follows directly from the structure of the function bodies and from Specification 11.8 (which implies that each call to RecycleBlockNS() at line 412 terminates in $H$).

For MiddleOpS(), it suffices to show that the busy-wait loop at line 419 terminates, since no other loops exist and by Specification 11.8 (which implies that each call to RecycleBlockS() at line 422 terminates in $H$). Suppose for contradiction that some execution of line 419 does not terminate. Then $p_{special}$ repeatedly reads $WaitFlagSpecial[q] = false$ at line 419 for some non-special process ID $q$. Note that $p_{special}$ previously read $ActiveSpecial[q] = true$ at line 418, and so at that point $q$ had completed line 413 of BeforeOpNS(), but had not subsequently completed line 414 of AfterOpNS(). Since $H$ is fair, $q$ eventually makes a call to AfterOpNS(), where it executes line 415 and assigns $WaitFlagSpecial[q] = true$. This follows because the ECAS implementations $I_{E-DSM}$ and $I_{EW-DSM}$ on which $I'_{E-DSM}$ and $I'_{EW-DSM}$ (respectively) are based satisfy Specification 6.2 (termination), and because termination is preserved during the transformation from $I_{E-DSM}/I_{EW-DSM}$ to $I'_{E-DSM}/I'_{EW-DSM}$ (see Theorems 11.12 and 11.20 and discussion in Section 12.3.1). Since only $p_{special}$ can assign $WaitFlagSpecial[q] = false$, and only at line 417 of MiddleOpS(), this contradicts $p_{special}$ repeatedly reading $WaitFlagSpecial[q] = false$ at line 419.

Theorem 12.17. For any history $H$ of $I'_{E-DSM}$ or $I'_{EW-DSM}$, each call to BeforeOpNS() and AfterOpNS() incurs $O(1)$ RMRs in the DSM model provided that each call to RecycleBlockNS() incurs $O(1)$ RMRs, and each call to MiddleOpS() incurs zero RMRs in the DSM model provided that each call to RecycleBlockS() incurs zero RMRs.

Proof. For BeforeOpNS() and AfterOpNS() (which are executed only by non-special processes), $O(1)$ RMR complexity follows directly from the structure of the function bodies and from our assumption on the RMR complexity of RecycleBlockNS(), which is called at line 412 of BeforeOpNS(). For MiddleOpS() (which is executed only by $p_{special}$), $O(1)$ RMR complexity follows from the structure of the function body, from the locality of $ActiveSpecial[1..N]$ and $WaitFlagSpecial[1..N]$ to $p_{special}$, and from our assumption on the RMR complexity of RecycleBlockS(), which is called at line 422.
Block Allocator

In this section we describe how to implement the special allocator. The special allocator consists of function $\text{AllocBlock}()$ for $p_{\text{special}}$ and two recycling functions: $\text{RecycleBlockS}()$ (for $p_{\text{special}}$) and $\text{RecycleBlockNS}()$ (for non-special processes).

The state of the special allocator is represented mostly in private variables of $p_{\text{special}}$. These include a queue $\text{FreshBlocks}$ of fresh blocks, a queue $\text{LastTwoBlocks}$ of blocks that may have a positive reference count, and a list $\text{DirtyBlocks}$ of blocks awaiting a reset. ($\text{DirtyBlocks}$ is not a queue because elements may be removed from any position.) These are analogous to $\text{FreshBlocks}$, $\text{RecentBlocks}$ and $\text{DirtyBlocks}$ from Section 11.3. Calls to $\text{AllocBlock}()$ and $\text{RecycleBlockS}()$ by $p_{\text{special}}$ move blocks among the special allocator’s data structures according to the pattern shown in Figure 12.4, while maintaining their number and distinctness (see Lemma 12.18). In each call to $\text{AllocBlock}()$, $p_{\text{special}}$ moves one block from $\text{FreshBlocks}$ to $\text{LastTwoBlocks}$ (at lines 429–430). In each call to $\text{RecycleBlockS}()$, $p_{\text{special}}$ moves one block from $\text{LastTwoBlocks}$ to $\text{DirtyBlocks}$ (at lines 432–433) and $O(N)$ blocks from $\text{DirtyBlocks}$ to $\text{FreshBlocks}$ (at lines 450–451).

![Figure 12.4: Possible transitions of special blocks among data structures of the special allocator.](image)

In addition to these private queues, there is a shared array $\text{ResetQueue}[1..N]$ of queues where $\text{ResetQueue}[q]$ is shared only by $p_{\text{special}}$ and $q$, and is local to $p_{\text{special}}$. (Such queues can be constructed as described in Chapter 10. As we show in the proof of Lemma 12.24, the length of each queue is $O(1)$ and so the state of the queue is sufficiently small to satisfy Condition 10.3.) The queue $\text{ResetQueue}[q]$ stores the addresses of blocks that are ready for $q$ to reset. It is a producer-consumer queue in the sense that $p_{\text{special}}$ enqueues elements inside $\text{RecycleBlockS}()$, and $q$ then dequeues them inside $\text{RecycleBlockNS}()$. It is possible for one block to be in multiple queues in $\text{ResetQueue}[1..N]$ in the event that
multiple non-special processes have accessed this block.

Finally, we introduce a block field: \( \text{Dirty}[1..N] \) is a Boolean array local to \( p_{\text{special}} \), all elements initialized to false. This array is written only by non-special processes and read only by \( p_{\text{special}} \). For block \( x \), a non-special process \( q \) sets \( x \triangleright \text{Dirty}[q] \) to true just after acquiring a strong reference to block \( x \). This occurs just after \( q \) makes a call to \text{getCurBlock()} with response \( x \), and is not shown in Figures 12.5–12.6 because it occurs outside of the block allocator. Process \( p_{\text{special}} \) relies on \( x \triangleright \text{Dirty}[1..N] \) inside \text{RecycleBlockS()} (line 435) to determine which non-special processes have accessed block \( x \), and to decide how to populate the queues \( \text{ResetQueue}[1..N] \) (line 436).

The pseudo-code for \text{AllocBlock()} and the recycling functions \text{RecycleBlockS()} and \text{RecycleBlockNS()} is presented below in Figures 12.5–12.6. In \text{AllocBlock()} and \text{RecycleBlockS()}, we assume without loss of generality that certain groups of pseudo-code statements (that access only the private queues \( \text{FreshBlocks}, \text{DirtyBlocks} \) and \( \text{LastTwoBlocks} \)) are executed atomically, as noted in the pseudo-code at lines 429–431, 432–433 and 450–451.

Function \text{AllocBlock()} simply removes one block from \( \text{FreshBlocks} \) and appends it to \( \text{LastTwoBlocks} \) at lines 429–430. It then returns this block. (The order of elements in \( \text{LastTwoBlocks} \) is important, which is why we append to it at line 430 and remove the head element at line 432, but in \( \text{FreshBlocks} \) the order is not important.)

In function \text{RecycleBlockS()}, \( p_{\text{special}} \) first tries to identify a recyclable block \( b \) at lines 432–433, and then determines which non-special processes must help reset this block by checking \( b \triangleright \text{Dirty}[1..N] \) at lines 434–438. If \( b \triangleright \text{Dirty}[q] = \text{true} \) for some non-special process \( q \) then \( p_{\text{special}} \) inserts \( b \) into \( \text{ResetQueue}[q] \) at line 436. Next, \( p_{\text{special}} \) tries to identify blocks in \( \text{DirtyBlocks} \) that are “almost fresh” meaning that every non-special process has already done its share of the work needed to reset the block (lines 439–453). To that end, \( p_{\text{special}} \) checks for each \( b \) in \( \text{DirtyBlocks} \) whether \( b \triangleright \text{Dirty}[1..N] \) contains all false values at lines 442–447. If so, then \( p_{\text{special}} \) makes \( b \) fresh by resetting any registers it has accessed in \( b \) at line 449, as well as the base object \( x \triangleright B \) (of non-writable ECAS type) in the case of a writable ECAS implementation, and then moves \( x \) from \( \text{DirtyBlocks} \) to \( \text{FreshBlocks} \) at lines 450–451.

In function \text{RecycleBlockNS()}, a non-special process \( q \) checks whether \( \text{ResetQueue}[q] \) is empty at line 423. If it is not then \( q \) obtains the head element of \( \text{ResetQueue}[q] \) at line 424, which is a special block block \( b \) inserted earlier by \( p_{\text{special}} \) at line 436 of \text{RecycleBlockS()}, and resets any registers \( q \) has accessed in \( b \) at line 425. Then, \( q \)
assigns $b \triangleright Dirty[q] = \text{false}$ at line 426 to notify $p_{\text{special}}$ that it has done its share of work in resetting $b$. (Process $p_{\text{special}}$ relies on this while executing lines 442–447 of RecycleBlockS().) Finally, $q$ removes $b$ from $ResetQueue[q]$ at line 427.

**Declarations**

**Shared variables:** (global)

$ResetQueue[1..N]$ – array of FIFO queues, initially empty, where $ResetQueue[q]$ is local to $p_{\text{special}}$ and shared only by $p_{\text{special}}$ and $q$ ($O(1)$ RMRs per operation for $q$)

**Shared variables:** (per-block)

$Dirty[1..N]$ – array of Boolean, local to $p_{\text{special}}$ and initially false

**Private variables:** (process $p_{\text{special}}$)

$FreshBlocks$ – FIFO queue of block addresses, initially contains $4N$ fresh special blocks

$DirtyBlocks$ – list of block addresses, initially empty

$LastTwoBlocks$ – FIFO queue of block addresses, initially contains two fresh special blocks different from those in $FreshBlocks$

$q$ – process ID or ⊥, uninitialized

$i$ – integer, uninitialized

$almostFresh$ – Boolean, uninitialized

**Private variables:** (per-process)

$b$ – block address or ⊥, initially ⊥

---

**Function** RecycleBlockNS() for non-special processes

423 if $ResetQueue[\text{PID}].\text{length}() > 0$ then

424 \hspace{1em} $b := ResetQueue[\text{PID}].\text{getHead}()$

425 \hspace{1em} reset any registers accessed by PID in block $b$ (except those underlying the base object $b \triangleright B$ in the writable ECAS implementation) back to their initial values

426 \hspace{1em} write $b \triangleright Dirty[\text{PID}] := \text{false}$

427 \hspace{1em} $b := ResetQueue[\text{PID}].\text{dequeueHead}()$

428 end

---

Figure 12.5: Block allocator for $p_{\text{special}}$ – part 1 (declarations and subroutine RecycleBlockNS()).

We now begin our analysis of the special block allocator. As in Section 11.3 we model executions of the allocator’s functions (this time AllocBlock(), RecycleBlockS() or RecycleBlockNS()) using a concurrent system $S$, and we show that histories of $S$ satisfy Specifications 11.7 and 11.8 provided that Conditions 11.9 (a), 11.11 (a) and
Function AllocBlock() for process $p_{\text{special}}$

// Note: lines 429–431 executed atomically.
429 $b := \text{FreshBlocks}\text{.enqueueHead()}$
430 LastTwoBlocks\text{.enqueueTail}(b)$
431 return $b$

Function RecycleBlockS() for process $p_{\text{special}}$

// Prepare the next block for recycling.
// Note: lines 432–433 executed atomically.
432 $b := \text{LastTwoBlocks}\text{.enqueueHead()}$
433 DirtyBlocks\text{.enqueueTail}(b)$
434 for each non-special process ID $q$ do
435  if read($b \triangleright \text{Dirty}[q]$) = true then
436    ResetQueue[q].enqueueTail($b$)
437  end
438 end

// Update queue of fresh and dirty blocks.
439 for $i$ from 1 to DirtyBlocks.length() do
440  $b := \text{DirtyBlocks}\text{.getElement}(i)$
441  almostFresh := true
442  for each non-special process ID $q$ do
443    if read($b \triangleright \text{Dirty}[q]$) = true then
444      almostFresh := false
445      break (to line 449)
446    end
447  end
448  if almostFresh = true then
449    reset any registers accessed by process $p_{\text{special}}$ in block $b$ back to their initial values, and call ($b \triangleright B$).Reset() in the case of the writable ECAS implementation
    // Note: lines 450–451 executed atomically.
450  $b := \text{DirtyBlocks}\text{.removeElement}(i)$
451  FreshBlocks\text{.enqueueTail}(b)$
452  end
453 end

Figure 12.6: Block allocator for $p_{\text{special}}$ – part 2 (subroutines AllocBlock() and RecycleBlockS()).
Chapter 12. Space Complexity in Locally-Accessible ECAS

12.10–12.12 hold. We also show that each call to \texttt{AllocBlock()}, \texttt{RecycleBlockS()} or \texttt{RecycleBlockNS()} incurs \(O(1)\) RMRs.

\textbf{Lemma 12.18.} For any history \(H\) of \(S\) where Condition 12.10 holds, and where \texttt{FreshBlocks} is non-empty at each execution of line 429 of \texttt{AllocBlock()}, the elements of the data structures \texttt{FreshBlocks}, \texttt{DirtyBlocks} and \texttt{LastTwoBlocks} are the addresses of special blocks. Furthermore, these elements are distinct and their number is \(4N + 2\).

\textbf{Proof.} Since the correctness properties under consideration are safety properties it suffices to consider finite \(H\). We proceed by induction on the length of the history \(H\).

\textbf{Basis:} \(|H| = 0\). At initialization, \texttt{FreshBlocks} contains \(4N\) special blocks, \texttt{LastTwoBlocks} contains two special blocks different from those in \texttt{LastTwoBlocks}, and \texttt{LastTwoBlocks} is empty.

\textbf{Induction step:} For any \(i \in \mathbb{N}\), suppose that the lemma holds when \(|H| = i\), and consider the case when \(|H| = i+1\). Note that \texttt{AllocBlock()} (of the special allocator) and \texttt{RecycleBlockS()} are the only places where \texttt{FreshBlocks}, \texttt{DirtyBlocks} and \texttt{LastTwoBlocks} are accessed. Since only \(p_{\text{special}}\) calls \texttt{AllocBlock()} by definition (in the special allocator) and since only \(p_{\text{special}}\) calls \texttt{RecycleBlockS()} by Condition 12.10 (a), it suffices to consider the following cases:

\textbf{Case A:} in step \(i + 1\), \(p_{\text{special}}\) executes (atomically) lines 429–430 of \texttt{AllocBlock()}.

In this case, step \(i + 1\) dequeues one element \(b\) from \texttt{FreshBlocks} (which we assume is not empty) at line 429 and enqueues \(b\) into \texttt{LastTwoBlocks} at line 430. This and the induction hypothesis imply that \(b\) is the address of a special block, and furthermore that at the end of \(H\) the elements of \texttt{FreshBlocks}, \texttt{LastTwoBlocks} and \texttt{DirtyBlocks} are distinct special blocks and their total number is \(4N + 2\).

\textbf{Case B:} in step \(i + 1\), \(p_{\text{special}}\) executes (atomically) lines 432–433 of \texttt{RecycleBlockS()}.

In this case, step \(i + 1\) dequeues one element \(b\) from \texttt{LastTwoBlocks} at line 432 and inserts it into \texttt{DirtyBlocks} at line 433. The fact that \texttt{LastTwoBlocks} is not empty at line 432 follows because \texttt{LastTwoBlocks} is not empty initially, because it is accessed only at line 430 of \texttt{AllocBlock()} and line 432 of \texttt{RecycleBlockS()}, and by Condition 12.10 (b). The rest of the analysis is analogous to Case A.

\textbf{Case C:} in step \(i + 1\), \(p_{\text{special}}\) executes (atomically) lines 450–451 of \texttt{RecycleBlockS()}.

In this case, step \(i + 1\) removes one element \(b\) from \texttt{DirtyBlocks} at line 450 and enqueues it into \texttt{FreshBlocks} at line 451. The fact that \texttt{DirtyBlocks} is not empty at line 450 follows from the structure of the loop at lines 439–453 and from the fact only \(p_{\text{special}}\)
executes \texttt{RecycleBlockS()} by Condition 12.10 (a). The rest of the analysis is analogous to Case A.

\textbf{Lemma 12.19.} For any history \( H \) of \( S \) where Conditions 11.11 (a), 12.10 and 12.12 hold, and where \( \text{FreshBlocks} \) is non-empty at each execution of line 429 of \texttt{AllocBlock()}, and for any special block \( x \), \( \text{ref}(x) > 0 \) implies \( x \in \text{LastTwoBlocks} \).

\textit{Proof.} Since the correctness property under consideration is a safety property it suffices to consider finite \( H \). We proceed by induction on the length of the history \( H \).

\textbf{Basis:} \( |H| = 0 \). At initialization, \( \text{ref}(x) = 0 \) holds for every special block \( x \) by our convention that the initial block is not special. Thus, the lemma holds trivially.

\textbf{Induction step:} For any \( i \in \mathbb{N} \), suppose that the lemma holds when \( |H| = i \), and consider the case when \( |H| = i + 1 \). Consider any special block \( x \). Since \( x \) is always an element of exactly one of \( \text{FreshBlocks} \), \( \text{DirtyBlocks} \) and \( \text{LastTwoBlocks} \) by Lemma 12.18 (given that \( \text{FreshBlocks} \) is non-empty at each execution of line 429 of \texttt{AllocBlock()}), it suffices to consider cases when step \( i + 1 \) increases \( \text{ref}(x) \) increases above zero (and show that \( x \in \text{LastTwoBlocks} \)) or dequeues \( x \) from \( \text{LastTwoBlocks} \) (and show that \( \text{ref}(x) = 0 \)).

\textbf{Case A:} step \( i + 1 \) increases \( \text{ref}(x) \) above zero. It follows from Condition 11.11 (a) that in step \( i + 1 \) a call to \texttt{AllocBlock()} returns \( x \). Furthermore, since we assume that \( x \) is special, \( p_{\text{special}} \) makes this call, and so during this call \( p_{\text{special}} \) executes lines 429–431 of \texttt{AllocBlock()} (atomically). In this case, \( p_{\text{special}} \) inserts \( x \) at the tail of \( \text{LastTwoBlocks} \) at line 430. This implies that \( x \in \text{LastTwoBlocks} \) in state \( H[i + 1] \), as wanted.

\textbf{Case B:} step \( i + 1 \) dequeues \( x \) from \( \text{LastTwoBlocks} \). Block \( x \) can be dequeued from \( \text{LastTwoBlocks} \) only at line 432 of \texttt{RecycleBlockS()}, which only \( p_{\text{special}} \) executes by Condition 12.10 (a). In particular, this occurs when \( p_{\text{special}} \) executes lines 432–433 (atomically), after a prior execution of lines 429–431 where \( p_{\text{special}} \) enqueued \( x \) into \( \text{LastTwoBlocks} \). Now let \( H' \) be the subhistory of \( H \) beginning just after \( p_{\text{special}} \) enqueues \( x \) into \( \text{LastTwoBlocks} \) at line 430 and ending just before \( p_{\text{special}} \) removes \( x \) from \( \text{LastTwoBlocks} \) at line 432. Suppose that the former step occurs during \( p_{\text{special}} \)'s \( j \)'th call to \texttt{AllocBlock()}. Then step \( i + 1 \) occurs during \( p_{\text{special}} \)'s \((j + 2)\)'nd call to \texttt{RecycleBlockS()}. To see this, first note that the queue \( \text{LastTwoBlocks} \) initially contains two elements, and is only accessed at line 430 of \texttt{AllocBlock()} and line 432 of \texttt{RecycleBlockS()}. Furthermore, only process \( p_{\text{special}} \) accesses \( \text{LastTwoBlocks} \) by Condition 12.10 (a), and does so in FIFO order by Condition 12.10 (b).
Since step $i + 1$ occurs during $p_{\text{special}}$’s $(j + 2)$’nd call to $\text{RecycleBlockS}()$, it follows from Condition 12.12 (b) that $ref(x)$ reaches zero in $H'$. After this point in $H'$, since $x \in \text{LastTwoBlocks}$ it follows from Lemma 12.18 (and our assumption that $\text{FreshBlocks}$ is non-empty at each execution of line 429) that $x$ is not simultaneously in $\text{FreshBlocks}$, and so $p_{\text{special}}$’s allocator does not return $x$ again. Thus, $ref(x)$ does not increase above zero again by Condition 11.11 (a), and so $ref(x) = 0$ until the end of $H'$. In particular, $ref(x) = 0$ when $p_{\text{special}}$ dequeues $x$ from $\text{LastTwoBlocks}$ at line 432, as wanted. \[\square\]

**Lemma 12.20.** For any history $H$ of $S$ where Conditions 11.11 (a), 12.10 and 12.12 hold, and where $\text{FreshBlocks}$ is non-empty at each execution of line 429 of $\text{AllocBlock}()$, and for any block $b$, any access to $b$ that occurs inside the special allocator (i.e., inside function $\text{AllocBlock}()$ for $p_{\text{special}}$, $\text{RecycleBlockS}()$ or $\text{RecycleBlockNS}()$) occurs while $ref(b) = 0$ and does not change the state of any shared object in $b$ to one different from the initial state for that object.

**Proof.** The only accesses to $b$ inside the allocator occur in the recycling functions: at line 449 of $\text{RecycleBlockS}()$ and at lines 425–426 of $\text{RecycleBlockNS}()$. Furthermore, any such access occurs while $b \in \text{DirtyBlocks}$. For the access at line 449 of $\text{RecycleBlockS}()$, this follows from the fact that $b$ is obtained from $\text{DirtyBlocks}$ at line 440, and the fact that only $p_{\text{special}}$ accesses $\text{DirtyBlocks}$. For the access at lines 425–426 of $\text{RecycleBlockNS}()$, first note that these lines are executed only by some non-special process $q$, which finds $b$ at the head of $\text{ResetQueue}[q]$ at line 424, and then assigns $b \triangleright \text{Dirty}[q] = \text{false}$ at line 426. Thus, $p_{\text{special}}$ enqueued $b$ into $\text{ResetQueue}[q]$ earlier at line 436, prior to which $b \in \text{DirtyBlocks}$ and $b \triangleright \text{Dirty}[q] = \text{true}$ held at line 435 (see also line 433). Subsequently, $b$ was not removed from $\text{DirtyBlocks}$ before $q$’s execution of line 426, because this can only occur at line 450 and only after $p_{\text{special}}$ reads $b \triangleright \text{Dirty}[q] = \text{false}$ at line 443, which can happen only after $q$ executes line 426.

Now since $b \in \text{DirtyBlocks}$ whenever $b$ is accessed internally in the block allocator (i.e., inside the recycling functions), it follows from Lemma 12.18 (and our assumption that $\text{FreshBlocks}$ is non-empty at each execution of line 429) that $b$ is not in $\text{LastTwoBlocks}$ at that point, and so $ref(b) = 0$ follows by Lemma 12.19. Finally, it follows directly from line 449 and lines 425–426 that an access to block $b$ at one of these lines returns the shared object accessed to its initial state. \[\square\]

**Lemma 12.21.** For any history $H$ of $S$ where Conditions 11.9 (a), 11.11 (a), 12.10 and 12.12 hold, for any non-special process $q$ and any for any special block $b$, if $b$ is
continuously in \( \text{DirtyBlocks} \) in some subhistory \( H' \) of \( H \) then no process assigns \( b \triangleright \text{Dirty}[q] = \text{true} \) in \( H' \).

Proof. Since \( b \) is continuously in \( \text{DirtyBlocks} \) in \( H' \), \( b \) is continuously not in \( \text{LastTwoBlocks} \) in \( H' \) by Lemma 12.18. Consequently, \( \text{ref}(b) = 0 \) holds continuously in \( H' \) by Lemma 12.19. Then by Condition 11.9 (a) and Lemma 12.20, no process makes an access to \( b \) in \( H' \) that changes the state of \( b \triangleright \text{Dirty}[q] \) to one different from the initial state, which is \text{false}. \( \square \)

Lemma 12.22. For any history \( H \) of \( S \) where Conditions 11.9 (a), 11.11 (a), 12.10 and 12.12 hold, any block in \( \text{FreshBlocks} \) is fresh.

Proof. Since the correctness property under consideration is a safety property it suffices to consider finite \( H \). We proceed by induction on the length of the history \( H \).

Basis: \( |H| = 0 \). At initialization, every special block is fresh.

Induction step: For any \( i \in \mathbb{N} \), suppose that the lemma holds when \( |H| = i \), and consider the case when \( |H| = i + 1 \). Consider any special block \( b \). It suffices to consider cases when step \( i + 1 \) enqueues \( b \) into \( \text{FreshBlocks} \), or accesses \( b \) while \( b \) is in \( \text{FreshBlocks} \).

Case A: step \( i + 1 \) enqueues block \( x \) into \( \text{FreshBlocks} \). In this case we must show that \( b \) is fresh. By the algorithm, step \( i + 1 \) can only occur in \( \text{RecycleBlockS}() \) when an element is moved from \( \text{DirtyBlocks} \) to \( \text{FreshBlocks} \) at lines 450–451. Furthermore, only \( p_{\text{special}} \) may execute this step by Condition 12.10 (a). Let \( H' \) be the subhistory of \( H \) from the point where \( p_{\text{special}} \) reaches line 442 prior to enqueuing \( b \) into \( \text{FreshBlocks} \), up to but not including (the atomic execution of) lines 450–451. Note that throughout \( H' \) block \( b \) is in \( \text{DirtyBlocks} \) and so no process assigns \( b \triangleright \text{Dirty}[q] = \text{true} \) in \( H' \) by Lemma 12.21. Since \( p_{\text{special}} \) completes the loop at lines 442–447 in \( H' \) without executing lines 444–445 (because it reaches lines 450–451), this implies that \( b \triangleright \text{Dirty}[1..N] \) contains all \text{false} values at the end of \( H' \). This implies that either \( q \) has never accessed block \( b \), or \( q \) has accessed \( b \), and the last such access before the end of \( H' \) was an execution of lines 425–426. This is because any access to \( b \) by \( q \) outside of the allocator begins with the assignment \( b \triangleright \text{Dirty}[q] = \text{true} \) (by our assumption on how \( \text{Dirty}[1..N] \) is accessed) any the only access to \( b \) by \( q \) inside the allocator occurs at lines 425–426, which ends with the assignment \( b \triangleright \text{Dirty}[q] = \text{false} \). Thus, at the end of \( H' \) any shared object in \( b \) (except \( B \) in the case of the writable ECAS implementation) whose state was changed from the initial state by a non-special process \( q \) has been reset back to its initial state through an execution of lines 425–426 of \( \text{RecycleBlockNS}() \) by \( q \). Furthermore, any
shared object in \( b \) modified by \( p_{\text{special}} \) has also been reset at line 449, including the base object \( B \) if applicable. Thus, \( b \) is fresh when \( p_{\text{special}} \) inserts it into \( \text{FreshBlocks} \) following \( H' \).

**Case B:** step \( i + 1 \) accesses block \( b \) while \( b \in \text{FreshBlocks} \). In this case we must show that \( b \) is fresh at the end of \( H \). Because block \( b \) is in \( \text{FreshBlocks} \) in state \( H[i] \), \( b \) is not in \( \text{LastTwoBlocks} \) in \( H[i] \) by Lemma 12.18, hence \( \text{ref}(b) = 0 \) in \( H[i] \) by Lemma 12.19. Thus, by Condition 11.9 (a) step \( i + 1 \) is an access to \( b \) that occurs inside the special allocator, namely during a call to \( \text{AllocBlock()} \) by \( p_{\text{special}} \), \( \text{RecycleBlockS()} \) or \( \text{RecycleBlockNS()} \). By Lemma 12.20, this access does not change the state of any shared object in \( b \) to one different from the initial state. Since \( b \) is fresh in \( H[i] \) by the induction hypothesis, this implies \( b \) is also fresh in \( H[i + 1] \).

**Lemma 12.23.** For any history \( H \) of \( S \) where Condition 12.10 holds, and where \( \text{FreshBlocks} \) is non-empty at each execution of line 429 of \( \text{AllocBlock()} \), \( \text{LastTwoBlocks} \) contains exactly two elements when \( p_{\text{special}} \) is at lines 439–453 of \( \text{RecycleBlockS()} \).

**Proof.** Initially, \( \text{LastTwoBlocks} \) contains two elements. Whenever \( \text{LastTwoBlocks} \) is accessed, this occurs at line 430 of \( \text{AllocBlock()} \), where an element is enqueued (since we assume that \( \text{FreshBlocks} \) is non-empty at each execution of line 429), and at line 432 of \( \text{RecycleBlockS()} \), where an element is dequeued. Consequently, only \( p_{\text{special}} \) accesses \( \text{LastTwoBlocks} \) by Condition 12.10 (a), and moreover \( p_{\text{special}} \) does so in FIFO order by Condition 12.10 (b). Thus, \( \text{LastTwoBlocks} \) contains two elements whenever \( p_{\text{special}} \) is between executions of line 430 and line 432, and three elements otherwise. This implies \( \text{LastTwoBlocks} \) contains two elements whenever \( p_{\text{special}} \) is at lines 439–453 of \( \text{RecycleBlockS()} \), as wanted.

**Lemma 12.24.** For any history \( H \) of \( S \) where Conditions 11.9 (a), 11.11 (a), 12.10 and 12.12 hold, and where \( \text{FreshBlocks} \) is non-empty at each execution of line 429 of \( \text{AllocBlock()} \), for any non-special process \( q \) there are at most four blocks in \( \text{DirtyBlocks} \) (in any state) for which \( \text{Dirty}[q] = \text{true} \).

**Proof.** Since the correctness property under consideration is a safety property it suffices to consider finite \( H \), and so let \( k = |H| \). For any \( 0 \leq i \leq k + 1 \), let \( D(q, i) \) denote the set of blocks in \( \text{DirtyBlocks} \) in state \( H[i] \) for which \( \text{Dirty}[q] = \text{true} \). (This set is well-defined because by Lemma 12.18 \( \text{DirtyBlocks} \) contains only special blocks in state \( H[i] \).) We must show that for any \( 0 \leq i \leq k + 1 \), \( |D(q, i)| \leq 4 \). To that end, we will first consider
another quantity: let \( Q(q, i) \) denote the set of blocks in \( \text{DirtyBlocks} \) in state \( H[i] \) for which \( \text{Dirty}[q] = \text{true} \) and which are also in \( \text{ResetQueue}[q] \). To complete the proof, it suffices to show that for any \( 0 \leq i \leq k + 1 \), \( |D(q, i)| \leq |Q(q, i)| + 1 \) and \( |Q(q, i)| \leq 3 \).

**Part 1:** \( |D(q, i)| \leq |Q(q, i)| + 1 \). It suffices to show that \( D(q, i) \) is a superset of \( Q(q, i) \) containing at most one additional element. First, note that while a special block \( b \) is in \( D(q, \ldots) \) or \( Q(q, \ldots) \), \( b \in \text{DirtyBlocks} \) holds and so no process assigns \( b \triangleright \text{Dirty}[q] = \text{true} \) by Lemma 12.21. Thus, if \( b \) enters \( D(q, \ldots) \) or \( Q(q, \ldots) \) this occurs when \( b \) is added to \( \text{DirtyBlocks} \) at line 433 of \( \text{RecycleBlockS}() \), or enqueued into \( \text{ResetQueue}[q] \) at line 436 of \( \text{RecycleBlockS}() \). Furthermore, any time \( b \) is enqueued into \( \text{ResetQueue}[q] \), it is first added to \( \text{DirtyBlocks} \) during the same call to \( \text{RecycleBlockS}() \). On the other hand, when \( b \) leaves \( D(q, \ldots) \) or \( Q(q, \ldots) \), then this happens as a result of \( q \) assigning \( b \triangleright \text{Dirty}[q] = \text{false} \) at line 426. To follows for two reasons: First, before \( q \) removes \( b \) from \( \text{ResetQueue}[q] \) at line 426, it first assigns \( b \triangleright \text{Dirty}[q] = \text{false} \) at line 427 after finding \( b \) at the head of \( \text{ResetQueue}[q] \) at line 424. Second, before \( p_{\text{special}} \) removes \( b \) from \( \text{DirtyBlocks} \) at line 450 of \( \text{RecycleBlockS}() \) (which only \( p_{\text{special}} \) calls by Condition 12.10 (a)), \( b \triangleright \text{Dirty}[q] = \text{false} \) holds by \( p_{\text{special}} \)'s prior execution of lines 440–449 for block \( b \), particularly due to the way \( \text{almostFresh} \) is used at lines 441, 444 and 448.

It follows from our analysis above that for any special block \( b \), exactly one of three cases applies in any state \( H[i] \):

1. \( b \) is in \( D(q, i) \) but not in \( Q(q, i) \) because it has been added to \( \text{DirtyBlocks} \) at line 433 of \( \text{RecycleBlockS}() \) and not yet added to \( \text{ResetQueue}[q] \) at line 436 of \( \text{RecycleBlockS}() \); or
2. \( b \) is in \( D(q, i) \) and in \( Q(q, i) \) because it has been added to \( \text{DirtyBlocks} \) at line 433 of \( \text{RecycleBlockS}() \) and added to \( \text{ResetQueue}[q] \) at line 436 of \( \text{RecycleBlockS}() \) and \( q \) has not yet assigned \( b \triangleright \text{Dirty}[q] = \text{false} \) at line 426; or
3. \( b \) is neither in \( D(q, i) \) nor in \( Q(q, i) \) because \( b \triangleright \text{Dirty}[q] = \text{false} \) or \( b \not\in \text{DirtyBlocks} \).

Since only \( p_{\text{special}} \) calls \( \text{RecycleBlockS}() \) by Condition 12.10 (a), it follows that at most one block is in the first category, and so \( D(q, i) \) is a superset of \( Q(q, i) \) containing at most one additional element, as wanted.

**Part 2:** \( |Q(q, i)| \leq 3 \). Suppose for contradiction that \( |Q(q, i)| \geq 4 \). It follows from the initialization of \( \text{ResetQueue}[q] \) that \( Q(q, 0) \) is empty. It also follows from our discussion in Part 1 (of how elements enter and leave \( Q(q, \ldots) \)) that \( |Q(q, j)| \) changes by at most one
in each step. Consequently, \(|Q(q, i)| \geq 4\) implies that \(|Q(q, j)| = 0\) and \(|Q(q, j')| = 4\) for some choice of \(j\) and \(j'\) where \(0 \leq j < j' \leq k\). Furthermore, without loss of generality we can choose \(j\) and \(j'\) so that \(1 \leq Q(q, l) \leq 3\) for \(j < l < j'\). Now let \(H'\) be the subhistory of \(H\) from step \(j + 1\) to step \(j'\). Let \(A\) be the number of elements added to \(Q(q, ...)\) in \(H'\), let \(R\) be the number of elements removed from \(Q(q, ...)\) in \(H'\). (In this context we count elements with multiplicity.) To derive a contradiction, it suffices to show that \(A - R < 4\) since we assume that \(|Q(q, ...)\) grows by four in \(H'\).

As explained in Part 1, whenever an element \(b\) is added to \(Q(q, ...)\), this happens at line 436 of \texttt{RecycleBlockS()}, which occurs after \(b\) is removed from \texttt{LastTwoBlocks} at line 432. By Lemma 12.23, there are at most two such elements in \texttt{LastTwoBlocks} at the beginning of \(H'\), and so at least \(A - 2\) of the \(A\) elements inserted into \texttt{ResetQueue}[q] in \(H'\) are first inserted into \texttt{LastTwoBlocks} after the beginning of \(H'\). Now any element \(b\) added to \texttt{LastTwoBlocks} after the beginning of \(H'\) is inserted at line 430 when \(b\) is allocated, and is fresh at that point because it is obtained from \texttt{FreshBlocks} (line 429) and by Lemma 12.22. Since \(b \triangleright Dirty[q] = \text{true}\) holds subsequently at line 435 before \(b\) is inserted into \texttt{ResetQueue}[q], this implies that \(q\) has accessed \(b\) between \(b\) being allocated and inserted into \texttt{ResetQueue}[q]. By Condition 11.9 (a) and Lemma 12.20, this means that \(q\) acquires a strong reference to a special block at least \(A - 2\) times in \(H'\), and so by Conditions 12.10 (c) and 12.12 (a) \(q\) makes at least \(A - 3\) complete calls to \texttt{RecycleBlockNS()} in \(H'\). Since \texttt{ResetQueue}[q] is never empty in \(H'\) after state \(H[j]\), it follows from lines 423–427 of \texttt{RecycleBlockNS()} that \(q\) executes line 426 at least \(A - 3\) times in \(H'\). This implies that \(q\) removes at least \(A - 3\) elements from \(Q(q, ...)\) in \(H'\) because each execution of line 426 changes \(x \triangleright Dirty[q]\) from \text{true} to \text{false} for some block \(x \in Q(q, ...)\). (To see why \(x \in Q(q, ...)\) immediately before such a step, note that \(Q(q, ...) \subseteq D(q, ...)\), as explained in Part 1. To see why \(x \triangleright Dirty[q] = \text{false}\) immediately before such a step, note that \(x \triangleright Dirty[q] = \text{true}\) when \(p_{\text{special}}\) enqueues \(x\) into \texttt{ResetQueue}[q], as explained earlier, and that after executing line 426 \(q\) dequeues \(x\) from \texttt{ResetQueue}[q] at line 427 before executing line 426 again.) Thus, \(R \geq A - 3\) holds, which implies \(A - R \leq 3\), as needed for contradiction.

\[\square\]

**Lemma 12.25.** For any history \(H\) of \(S\) where Conditions 11.9 (a), 11.11 (a), 12.10 and 12.12 hold, whenever \(p_{\text{special}}\) is about to remove an element from \texttt{FreshBlocks} at line 429 of \texttt{AllocBlock()}, \texttt{FreshBlocks} is not empty.

**Proof.** Suppose for contradiction that the lemma is false and consider the shortest history
$H'$ at the end of which $p_{\text{special}}$ is at line 429 with $FreshBlocks$ empty. It follows from Condition 12.10 (b) that $p_{\text{special}}$ called $\text{RecycleBlockS}()$ after its last call to $\text{AllocBlock}()$, where it removed the last element from $FreshBlocks$, and so $FreshBlocks$ is empty during $p_{\text{special}}$'s execution of lines 439–453 of that call to $\text{RecycleBlockS}()$.

Let $H''$ be the subhistory of $H'$ corresponding to this execution of lines 439–453. We will now show that $DirtyBlocks$ contains exactly $4N$ elements throughout $H''$, using the fact that $H'$ satisfies the hypotheses of Lemmas 12.18–12.24. Recall that $FreshBlocks$ contains $4N$ blocks initially and yet $FreshBlocks$ is empty in $H''$ as explained earlier, $LastTwoBlocks$ contains exactly two elements at initialization and also in $H''$ by Lemma 12.23, and the total number of elements in $FreshBlocks$, $DirtyBlocks$ and $LastTwoBlocks$ is constant in $H''$ by Lemma 12.18. Since $FreshBlocks$ contains $4N$ fewer blocks in $H''$ than initially, and since $LastTwoBlocks$ contains the same number of blocks in $H''$ as initially, it follows that $DirtyBlocks$ contains $4N$ more blocks in $H''$ than initially, which is $4N$, as wanted.

Now let $H'''$ denote the subhistory of $H''$ corresponding to $p_{\text{special}}$’s execution of the loop at lines 442–447 of $\text{RecycleBlockS}()$ for $q$’s ID. Note that by Lemma 12.24, at any point in $H'''$ for any non-special process $p$ there are at most four blocks in $DirtyBlocks$ for which $\text{Dirty}[p] = \text{true}$. Since the state of $DirtyBlocks$ does not change in $H'''$ by the algorithm and Condition 12.10 (a), it follows from Lemma 12.21 that $x \triangleright \text{Dirty}[q] = \text{false}$ holds continuously in $H'''$ for at least $4N-4(N-1) = 4$ blocks because there are $N-1$ non-special processes and exactly $4N$ blocks in $DirtyBlocks$ in $H'''$. Thus, as $p_{\text{special}}$ executes lines 441–447 with $b$ equal to the address of such a block, $p_{\text{special}}$ does not execute lines 444–445, and so it later enqueues $b$ into $FreshBlocks$ at line 451 before returning from $\text{RecycleBlockS}()$. Since $p_{\text{special}}$ does not remove any element from $FreshBlocks$ subsequently, this contradicts $FreshBlocks$ being empty at the end of $H''$.

\textbf{Theorem 12.26.} For any history of $S$ where Conditions 11.9 (a), 11.11 (a), 12.10–12.12 hold, Specifications 11.7–11.8 also hold. Furthermore, every call by $p_{\text{special}}$ to $\text{AllocBlock}()$ and $\text{RecycleBlockS}()$ incurs zero RMRs in the DSM model, and every call by a non-special process to $\text{RecycleBlockNS}()$ incurs $O(1)$ RMRs in the DSM model.

\textit{Proof.} Specification 11.7 (a) follows from the fact that any element returned to $p_{\text{special}}$ by $\text{AllocBlock}()$ is obtained from $FreshBlocks$ at line 429, and so it is fresh by Lemmas 12.22 and 12.25. It also has reference count zero just before being returned by Lemmas 12.18, 12.19 and 12.25, and so it has reference count one just after being re-
turned. Specification 11.7 (b) follows from Lemmas 12.20 and 12.25. Specification 11.8 follows directly from the structure of the functions AllocBlock(), RecycleBlockS() and RecycleBlockNS(). (For the statements at line 449 of RecycleBlockS() and line 425 of RecycleBlockNS(), note that each block contains a finite number of registers.)

The RMR complexity of AllocBlock() and RecycleBlockS() to \(p_{special}\) follows from the locality of the shared objects ResetQueue[1..N] and block fields Dirty[1..N] to \(p_{special}\), and the fact that \(p_{special}\) incurs zero RMRs at line 449 by Condition 12.11. The RMR complexity of RecycleBlockNS() for non-special processes follows from the structure of this function and from the fact that a process incurs \(O(1)\) RMRs at line 425 by Condition 12.11.

**Theorem 12.27.** The special block allocator uses \(4N + 2\) blocks.

*Proof.* It suffices to count the number of blocks at initialization: \(4N\) are in FreshBlocks, 2 are in LastTwoBlocks and DirtyBlocks is empty. □

**Combining the Non-Special and Special Allocators**

In this section we state the correctness properties of the block allocator for ECAS implementations that satisfy the locality property in the DSM model. This allocator comprises functions AllocBlock() and RecycleBlock() of the non-special allocator, which are called only by non-special processes (see Section 12.3.1); functions AllocBlock() and RecycleBlockS() of the special allocator, which are called only by the special process (see Section 12.3.2); and function RecycleBlockNS() of the special allocator, which is called only by non-special processes (see Section 12.3.2). The functions RecycleBlock(), RecycleBlockS() and RecycleBlockNS() are the recycling functions (see Definition 11.6).

(To complete the picture, recall also that several functions are needed to instrument the ECAS implementation in support of various conditions. Non-special processes call BeforeOpNS() and BeforeOp() at the beginning of each operation execution, and then AfterOp() and AfterOpNS() at the end. Non-special processes also call MiddleOp(\(x, y\)) shortly after a visible allocation, and \(p_{special}\) calls MiddleOpS() shortly after any allocation.)

To state the correctness properties of the allocator for the DSM model, we model executions of AllocBlock() and the recycling functions as a concurrent system \(S\), and
we show that histories of $S$ satisfy Specifications 11.7 and 11.8 provided that Conditions 11.9–11.11 (amended as described in Section 12.3.1) and Conditions 12.10–12.12 hold. We also state the RMR complexity of calls to $\text{AllocBlock}()$ and the recycling functions.

**Theorem 12.28.** For any history $H$ of $S$ where steps of non-special processes satisfy Conditions 11.9–11.11 (amended as described in Section 12.3.1), and steps of all processes satisfy Conditions 11.9 (a), 11.11 (a) and 12.10–12.12, Specifications 11.7–11.8 also hold. Furthermore, every call by $p_{\text{special}}$ to $\text{AllocBlock}()$ and $\text{RecycleBlockS}()$ incurs zero RMRs in the DSM model, and every call by a non-special process to $\text{AllocBlock}()$, $\text{RecycleBlock}()$ or $\text{RecycleBlockNS}()$ incurs $O(1)$ RMRs in the DSM model.

**Proof.** For functions $\text{AllocBlock}()$ and $\text{RecycleBlock}()$ of the non-special allocator, Specifications 11.7–11.8 hold by Theorem 12.8. Each call to $\text{AllocBlock}()$ or $\text{RecycleBlock}()$ by a non-special process incurs $O(1)$ RMRs by Theorem 12.8. For functions $\text{AllocBlock}()$, $\text{RecycleBlockS}()$ and $\text{RecycleBlockNS}()$ of the non-special allocator, Specifications 11.7–11.8 hold by Theorem 12.26. Each call to $\text{AllocBlock}()$ or $\text{RecycleBlockS}()$ by $p_{\text{special}}$ incurs zero RMRs by Theorem 12.26. Each call to $\text{RecycleBlockNS}()$ by a non-special process incurs $O(1)$ RMRs by Theorem 12.26.

**Theorem 12.29.** The block allocator for the DSM model uses $7N + 8$ blocks.

**Proof.** The non-special allocator uses $3N + 6$ blocks by Theorem 12.9. The special allocator uses $4N + 2$ blocks by Theorem 12.27. The total number of blocks is $7N + 8$.

### 12.4 Conclusion

In this chapter we showed how to adapt our memory management scheme from Chapter 11 so that it can be used with our implementations from Chapter 7 without breaking their locality properties. (The implementations from Chapter 10 can use the scheme from Chapter 11 directly, despite being locally-accessible, because in those implementations only non-special processes access blocks or the block manager.) The space complexity bounds for the locally-accessible implementations with memory management are the same as stated at the end of Chapter 11 for non-locally-accessible implementations.

Having presented all our implementations and shown how to bound space complexity, we will close the thesis with a discussion of open problems and future work in Chapter 13.
Chapter 13

Open Problems and Future Work

In this thesis we proved three principal results. First, we showed that CAS and LL/SC are no stronger than reads and writes alone under a ranking that defines the strength of a primitive as the RMR complexity of implementing it from reads and writes only. We did so by presenting $O(1)$-RMR implementations of CAS and LL/SC from reads and writes only. Second, we strengthened our $O(1)$-RMR implementations of CAS and LL/SC with locality properties that allow these implementations to simulate their hardware counterparts not only in terms of linearizability, but also in terms of RMR complexity. We used these locally-accessible implementations to establish our third principal result, which is that any algorithm based on reads, writes, CAS and LL/SC can be transformed so that it uses reads and writes only (by replacing objects on which CAS and LL/SC are applied with our implementations) in a way that introduces at most a constant-factor increase in RMR complexity and also preserves important correctness properties.

We leave open several interesting problems related to our principal results. To begin with, in our proof of the first result we defined the synchronization primitives CAS and LL/SC as shared objects supporting certain operation types, and we assumed that a single shared object would not be accessed using both CAS and LL/SC operations. This assumption is reasonable since multiprocessors typically support either CAS or LL/SC, and not both. It is interesting to ask what would be the specification of a shared object that supports both CAS and LL/SC. We addressed this question partly by defining the ECAS type in Chapter 6, which provides an operation type ECAS that can behave either like CAS or like SC. Unfortunately, for technical reasons our implementation of ECAS is linearizable only if ECAS is called in a consistent manner – in one history, either all calls to ECAS simulate CAS or all calls simulate SC. This restriction is denoted by Condition 6.4
in our analysis. We see no reason why an $O(1)$-RMR implementation of ECAS from reads and writes without this restriction would be impossible, but we also do not know how such an implementation could be obtained using the particular approach presented in this thesis.

Another open problem pertains to a potential practical application of our results. Since in practice CAS is a slower machine instruction than a read or write, it is natural to ask whether it is possible to beat the performance of a hardware CAS instruction using a software implementation of CAS based on reads and writes. As regards atomic reads and writes, it seems unlikely that this can be done in a modern multiprocessor because the performance gap between CAS and other atomic instructions is quite small. The gap is considerably larger between CAS and non-atomic reads and writes, however. We leave open the problem of how to model formally the particular non-atomic reads and writes provided by modern multiprocessors, and how such instructions can be used to simulate CAS efficiently.

Since our implementations are in some cases quite complex, further work arising from this thesis could look at how to simplify these implementations conceptually. This could in turn improve their performance as well. There are three main sources of complexity at hand in this thesis: asynchrony, the use of weak base objects (i.e., atomic read/write registers), and our various RMR complexity requirements. Since asynchrony and read/write registers are inherent in the research problem at hand, our greatest hope for reducing complexity lies in weaker RMR complexity requirements. As we remarked at the end of Chapter 7, worst-case $O(1)$ RMR complexity per operation is not necessary for proving our third principal result, and similarly our locality properties in the CC model (see Definitions 7.2 and 7.3) are stronger than necessary for proving RMR preservation (see Definition 7.1). We leave open the question of how the RMR preservation property can be achieved using weaker worst-case RMR complexity and locality properties. Another interesting possibility, raised by Prof. Sam Toueg, is to relax the RMR preservation property itself by looking at the total number of RMRs performed in an entire history by all processes rather than by each process individually.

Once specific way to simplify our implementations given weaker RMR complexity requirements is to simplify the memory management scheme presented in Chapters 11–12. Consider the subproblems of waiting until the reference count for a block reaches zero, and hard-resetting a block. Each task requires a super-constant number of RMRs in our scheme, in the first case because we must wait for many processes, and in the second case
because each block contains many shared objects. The $O(1)$-RMR requirement forces us to perform each task incrementally, which introduces overhead because each task must be “saved” and “restored” as processes take turns completing it (in certain successful ECAS operation executions). An alternate approach, suggested by Prof. Vassos Hadzilacos, is for each process to participate in memory recycling (i.e., call \texttt{RecycleBlock()}) less often and perform more work each time, while keeping the amortized cost of \texttt{RecycleBlock()} at $O(1)$ RMRs. For example, suppose that each process calls \texttt{RecycleBlock()} once every $\Theta(N^2 \log N)$ RMRs. In that case we can allow each call to \texttt{RecycleBlock()} to incur $\Theta(N^2 \log N)$ RMRs, which is enough to hard-reset an entire block (in the case of the non-writable implementation), and also to acquire and release a mutex for synchronization with other processes that might be executing \texttt{RecycleBlock()} concurrently. The disadvantage of this approach is that the total number of blocks needed grows from $\Theta(N)$ to $\Theta(N^2 \log N)$ because each process can allocate $\Theta(N^2 \log N)$ blocks before calling \texttt{RecycleBlock()}. (In our current approach a process can make at most one “visible” allocation between calls to \texttt{RecycleBlock()}.)}
Bibliography


