LOCAL-SPIN ABORTABLE MUTUAL EXCLUSION

by

Hyonho Lee

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

Copyright © 2011 by Hyonho Lee
Abstract

Local-spin Abortable Mutual Exclusion

Hyonho Lee

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2011

Abortable mutual exclusion is a variant of mutual exclusion, in which processes are allowed to abort in the trying protocol. Scott presented the first local-spin abortable mutual exclusion algorithms. They are based on a queue and perform $O(1)$ remote memory accesses (RMAs) when no processes abort. However, they use unbounded space and a process can perform an unbounded number of RMAs, in the worst case, to enter the critical section. The only other local-spin abortable mutual exclusion algorithm is by Jayanti. It has bounded space and RMA complexity, but a process always performs $\Theta(\log N)$ RMAs to enter the critical section, where $N$ is the number of processes.

In this thesis, three new, bounded space, abortable mutual exclusion algorithms are presented. We give the first local-spin abortable mutual exclusion algorithm that uses only registers. It has $\Theta(\log N)$ RMA complexity, even if no processes abort. We also present the first local-spin abortable mutual exclusion algorithm using more general primitives which has bounded space and in which each process performs $O(1)$ RMAs to enter the critical section when no processes abort. However, this algorithm is local-spin only for a certain type of cache-coherent model. Finally, we present an abortable mutual exclusion algorithm which is local-spin in any cache-coherent model and in which each process performs $O(1)$ RMAs to enter the critical section when no processes abort. We develop a new reference counting method to bound the space used in this algorithm.
Acknowledgements

First, I would like to thank my supervisor, Professor Faith Ellen. I owe a lot to her. She guided all aspects of my research. She taught me how to find problems, how to read papers, and how to write papers. Moreover, she taught me to think critically and to write clearly and precisely. Most of all, she always encouraged me whenever I struggled.

I would like to thank my committee members, Professor Vassos Hadzilacos and Professor Angela Demke-Brown, and my external examiner, Dr. Maged Michael. They gave me numerous invaluable comments and feedback on my work.

I thank my colleague, Robert Danek, for collaborating with me on our paper about abortable mutual exclusion algorithms using only registers (Chapter 4). He converted my algorithm into an FCFS abortable algorithm, which can be applied to group mutual exclusion and $k$--exclusion. We also discussed many problems about distributed computing and exchanged ideas with each other.

My family always supported me while I was in graduate school. I thank my wife, Heeseung Yoo and my three children, Han, Louise, and Luke. They were patient and always encouraged me with love. I also thank my mother, Sang Mee Lee, who gave me an opportunity to study in a great university.

Finally, I would like to thank my father, Yong Seo Lee, who passed away two years ago. He was a great father and wanted me to obtain a doctoral degree. I would like to dedicate this thesis to my father who watches over me in heaven.
Contents

1 Introduction 1

2 Preliminaries 6
  2.1 Shared variables and atomic operations 6
  2.2 System models 10
  2.3 The mutual exclusion problem and its variants 14
  2.4 Local-spin algorithms and RMA complexity 19

3 Related Work 21
  3.1 Local-spin mutual exclusion algorithms 22
    3.1.1 Tree-based local-spin mutual exclusion algorithms 26
    3.1.2 Queue-based local-spin mutual exclusion algorithms 32
  3.2 Lower bounds for mutual exclusion 37
  3.3 Local-spin abortable mutual exclusion algorithms 38
  3.4 $k$-exclusion and group mutual exclusion 40
  3.5 Memory reclamation methods 41
    3.5.1 Overview and definitions 41
    3.5.2 Garbage collection 43
    3.5.3 Hazard pointers 44
    3.5.4 Reference counting 45
    3.5.5 Lock-free reference counting using DCAS 46
    3.5.6 Lock-free reference counting using a free-list 48
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.7</td>
</tr>
<tr>
<td>3.5.8</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4.1</td>
</tr>
<tr>
<td>4.2</td>
</tr>
<tr>
<td>4.3</td>
</tr>
<tr>
<td>4.4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>5.1</td>
</tr>
<tr>
<td>5.2</td>
</tr>
<tr>
<td>5.3</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6.1</td>
</tr>
<tr>
<td>6.2</td>
</tr>
<tr>
<td>6.3</td>
</tr>
<tr>
<td>6.4</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>7.1</td>
</tr>
<tr>
<td>7.2</td>
</tr>
<tr>
<td>7.3</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8.1</td>
</tr>
<tr>
<td>8.2</td>
</tr>
<tr>
<td>8.3</td>
</tr>
<tr>
<td>8.4</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the mutual exclusion problem [Dij65], multiple processes try to use a shared resource that cannot be simultaneously accessed. In a mutual exclusion algorithm, processes perform a trying protocol before accessing the resource and then perform an exit protocol after accessing the resource. When a process accesses the shared resource, it is considered to be in the critical section. When a process does other work not related to the use of the resource, it is considered to be in the remainder section. A process that invokes the trying protocol must wait while another process is in the critical section. Although the process in the critical section eventually finishes the critical section, it may stay in the critical section for an unbounded amount of time. Hence, processes may wait in the trying protocol for a very long time.

Abortable mutual exclusion [SI01] is a variant of classical mutual exclusion, in which a process in the trying protocol is allowed to stop waiting for the critical section to become available, by performing an abort protocol, which returns the process to the remainder section within a bounded number of steps. Abortable mutual exclusion can be useful in real-time applications or in database systems, because, in these systems, users may want to abort any operation that takes too long [SI01].

Processes perform the exit protocol after finishing the critical section, so they perform
the exit protocol in the same order as they enter the critical section. However, processes can abort anytime during waiting in the trying protocol. Hence, the order in which processes abort can be arbitrary and more than one process may perform the abort protocol concurrently. Consequently, allowing aborts might make the mutual exclusion problem more difficult.

In shared memory models, processes communicate with each other only via shared variables, so waiting processes must keep reading shared variables to find out that the critical section is available. Such busy-waiting may cause processes to perform an unbounded number of steps during the trying protocol. In the distributed shared memory (DSM) and cache-coherent (CC) models, the cost for a process to access its own shared memory or cache is often much less than the cost to access memory located remotely. Hence, in these models, counting only remote memory accesses (RMAs) is a good measure of the time complexity of an algorithm. To achieve a bounded number of RMAs, many papers about mutual exclusion have considered local-spin algorithms. In such algorithms, each process accesses only its local memory or cache during busy-waiting, except for a bounded number of remote memory accesses each time it busy-waits.

The simplest mutual exclusion algorithm is a try-lock, in which a process in the trying protocol repeatedly performs test_and_set on a variable until it returns TRUE. When a process finishes the critical section, it resets the variable. This algorithm is not local-spin and processes may be locked out. But a process can abort by immediately returning to the remainder section, without performing any more test_and_set operations.

In some classical mutual exclusion algorithms, such as the Bakery algorithm [Lam74], which are not local-spin, each waiting process repeatedly checks the state of every other process. Then a process can abort by simply announcing that it is no longer trying. In such algorithms, there is no race condition between an exiting process and an aborting process.

Converting a local-spin mutual exclusion algorithm to a local-spin abortable mutual
exclusion algorithm is more complicated. In most local-spin algorithms, each waiting process waits for a different process. An exiting process tells the process which is waiting for it that it has finished the critical section. If a process $p$ is told after it already aborted, then the exiting process must tell the process waiting for $p$, if any, to enter the critical section. If process $p$ aborts after it is told, then $p$ must tell the process waiting for it to enter the critical section. If $p$ and the exiting process both tell the waiting process to enter the critical section at different times, then mutual exclusion can be violated: Consider the case when processes $p$ and $q$ tell process $r$ to enter the critical section at different times. Suppose that $p$ tells $r$ to enter the critical section, $r$ enters and finishes the critical section, $r$ re-enters the trying protocol and waits for another process $s$. If $q$ tells $r$ to enter the critical section while $s$ is in the critical section, then both $r$ and $s$ can be in the critical section, violating mutual exclusion.

Scott [Sco02] presented the first local-spin abortable mutual exclusion algorithms. They are constructed from fetch_and_store and compare_and_swap objects. His algorithms use unbounded space and there is no bound on the number of RMAs a process may perform to enter the critical section. Jayanti [Jay03] presented the first local-spin abortable mutual exclusion algorithm that uses a bounded number of shared variables. His algorithm is constructed from registers and load-linked/store-conditional (ll/sc) objects and uses unbounded sequence numbers, which require registers of unbounded size. Each process performs $\Theta(\log N)$ RMAs to enter the critical section in Jayanti’s algorithm, where $N$ is the number of processes. Hence, in the worst case, each process performs fewer RMAs to enter the critical section in Jayanti’s algorithm than in the Scott’s algorithms. However, when no processes abort, each process still performs $\Theta(\log N)$ RMAs to enter the critical section in Jayanti’s algorithm, but each process performs only $O(1)$ RMAs to enter the critical section in Scott’s algorithms.

A natural question is whether there is a local-spin abortable mutual exclusion algorithm with bounded space and RMAs in which each process performs $O(1)$ RMAs to
Chapter 1. Introduction

Table 1.1: Local-spin abortable mutual exclusion algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Atomic operations used besides READ and WRITE</th>
<th>Local-spin</th>
<th>RMA per passage if no aborts</th>
<th>worst case RMAs per passage</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scott [Sco02]</td>
<td>FETCH_AND_STORE, COMPARE_AND_SWAP</td>
<td>CC and DSM</td>
<td>O(1)</td>
<td>unbounded</td>
<td>unbounded</td>
</tr>
<tr>
<td>Jayanti [Jay03]</td>
<td>LL_SC</td>
<td>CC and DSM</td>
<td>(\Theta(\log N))</td>
<td>(\Theta(\log N))</td>
<td>(\Theta(N))</td>
</tr>
<tr>
<td>New Algorithm 1</td>
<td>-</td>
<td>CC and DSM</td>
<td>(\Theta(\log N))</td>
<td>(\Theta(\log N))</td>
<td>(\Theta(N))</td>
</tr>
<tr>
<td>New Algorithm 2</td>
<td>FETCH_AND_STORE, COMPARE_AND_SWAP</td>
<td>CC with cache-update</td>
<td>O(1)</td>
<td>(\Theta(N))</td>
<td>(\Theta(N))</td>
</tr>
<tr>
<td>New Algorithm 3</td>
<td>FETCH_AND_STORE, FETCH_AND_ADD</td>
<td>CC</td>
<td>O(1)</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(N^2))</td>
</tr>
</tbody>
</table>

enter the critical section when no processes abort. Such an algorithm would have better performance in systems, where aborts rarely happen.

Existing local-spin abortable mutual exclusion algorithms use powerful atomic operations. Hence, another natural question is whether there exists a local-spin abortable mutual exclusion algorithm using only registers. This thesis answers these questions by presenting three new algorithms. Table 1.1 summarizes our results and compares them with Scott’s and Jayanti’s algorithms.

Our first algorithm uses only registers and bounded space. It is a surprisingly simple modification of Yang and Anderson’s algorithm [YA95]. This is the first local-spin abortable mutual exclusion algorithm using only registers and the first local-spin abortable mutual exclusion algorithm using bounded space. In particular, it does not use unbounded sequence numbers.

Danek [Dan11, DL08] gave a transformation from any abortable mutual exclusion algorithm to a first-come-fist-served (FCFS) abortable mutual exclusion algorithm. Danek and Hadzilacos [DH04] gave a transformation to group mutual exclusion (defined in Section 2.3). Both transformations use only registers. Combined with our first algorithm, these transformations give the first FCFS local-spin abortable mutual exclusion algorithm using only registers and the first group mutual exclusion algorithm using only registers.

Our second algorithm is the first local-spin abortable mutual exclusion algorithm
with bounded space and RMAs, in which each process performs $O(k)$ RMAs to enter the critical section when only $k$ processes abort. In particular, it performs $O(1)$ RMAs when no process aborts. It uses only \texttt{compare\_and\_swap}, \texttt{fetch\_and\_store}, \texttt{read} and \texttt{write}. However, this algorithm is local-spin only in a restricted version of the CC model.

Our third algorithm is also an abortable mutual exclusion algorithm with bounded space and RMAs. Each process performs $O(k^2)$ RMAs to enter the critical section when only $k$ processes abort. This algorithm is local-spin in the CC model. It is constructed from a new shared object type, \textit{S-HAD}, which is a sequence that supports Head, Append, and Delete operations. We give two wait-free implementations of an S-HAD object. Our first implementation has $O(N^2)$ RMA complexity but uses unbounded space. Then, we extend it, using a generalization of reference counts, to achieve $O(N^2)$ space complexity, as well. Our new memory reclamation method is wait-free and very efficient in terms of RMAs. It uses only \texttt{test\_and\_set}, \texttt{fetch\_and\_add}, \texttt{fetch\_and\_store}, \texttt{read}, and \texttt{write} on $O(\log N)$ bit words, and each process performs $O(1)$ RMAs for recycling a record.

The rest of the thesis is organized as follows. Chapter 2 formally defines the problems and models we consider. Related work is presented in Chapter 3. Our first algorithm, using only registers, is presented and proved correct in Chapter 4. In Chapter 5, we present our algorithm for a restricted CC model. In Chapter 6, S-HAD is defined and a local-spin abortable mutual exclusion algorithm using S-HAD is presented. In Chapters 7 and 8, our implementations of S-HAD are presented and proved correct. Finally, Chapter 9 presents open problems.
Chapter 2

Preliminaries

2.1 Shared variables and atomic operations

We consider $N$ process asynchronous shared memory systems without failures. In such systems, processes communicate with each other using shared objects (also called shared variables) located in shared memory area. In contrast to private variables, which are accessible to only one process, shared objects can be accessed by any process. In this thesis, to distinguish between shared variables and private variables, an uppercase letter is used for the first letter of a shared variable, and a lowercase letter is used for the first letter of a private variable.

Each shared object has a set of possible values and a set of possible atomic operations that can be performed on it. An operation returns a response to the process that performed the operation and may update the value of the object. The initial value of a shared object is specified by algorithms that use it.

The most common operations are READ and WRITE. An object that supports only READ and WRITE is called a register. In this thesis, the following operations, defined in Figure 2.1, are also considered. Note that, in these definitions, the pseudo-code is executed atomically.
A test-and-set operation on a shared variable $V$ sets the value of $V$ to 1 and returns true if $V = 0$, and it returns false, otherwise. A reset operation sets the value of $V$ back to 0. \textsc{compare-and-swap} (or \textsc{cas}) takes two arguments in addition to the shared variable to which it is applied. \textsc{compare-and-swap}(\(V, x, y\)) returns false if $x$ differs from the current value of $V$. Otherwise, it sets the value of $V$ to $y$ and returns true.

A comparison-based operation changes the value of a shared variable only when it has a certain value. A comparison-based operation can be defined as follows, where \(\phi\) and \(\psi\) are arbitrary functions.

\[
\text{\textsc{compare} }& \phi \psi \ (V, x, y) \{
\begin{align*}
\ & \text{temp} := V \\
\ & \text{if } \text{temp} = x \text{ then } V := \phi(x,y) \ \text{fi} \\
\ & \text{return } \psi(\text{temp}, x, y) \}
\end{align*}
\]

\textsc{test-and-set} and \textsc{compare-and-swap} are examples of comparison-based operations.

\textsc{compare-and-swap} is convenient if a process wants to update a shared variable if and only if the shared variable has not been changed by another process. However, using a \textsc{compare-and-swap} operation, a process cannot distinguish between the case when a shared variable has not been changed and the case when it was changed and then changed back to its previous value. This is called the \textit{ABA problem}.

The ABA problem can be solved using \textit{tagging}: A shared variable has a counter and each time it is modified, the counter is incremented. However, each shared variable now has unbounded size. In practice, it suffices to use a large bounded counter.

\textsc{load-linked} (\textsc{ll}) and \textsc{store-conditional} (\textsc{sc}) operations solve the ABA problem without such disadvantages of tagging. \textsc{ll}(\(V\)) returns the value of the shared variable $V$, so \textsc{ll} behaves the same as \textsc{read}. \textsc{sc}(\(V, x\)) performed by process $p$ succeeds if and only if no process has performed a successful \textsc{sc} on $V$ since $p$’s latest \textsc{ll} on $V$. If \textsc{sc} succeeds,
then it changes $V$’s value to $x$ and returns $\text{TRUE}$. Otherwise, $V$’s value is unchanged and $\text{sc}$ returns $\text{FALSE}$. In the definition of an $\text{LL/sc}$ object presented in Figure 2.1, a Boolean array of size $N$, $X_V$, is used for each object, $V$. Initially $X_V$ is all $\text{FALSE}$, and $X_V[i]$ is $\text{TRUE}$ if and only if process $i$ performed $\text{LL}$ and no process has performed a successful $\text{sc}$ on $V$ since process $i$ last performed $\text{LL}$. When process $i$ performs $\text{LL}$ on $V$, it sets $X_V[i]$ to $\text{TRUE}$. When process $i$ performs $\text{sc}$ on $V$, it first checks if $X_V[i]$ is $\text{TRUE}$. If it is, it resets $X_V$ and updates $V$.

$\text{fetch\_and\_store}$ (also called $\text{swap}$) applied to a shared variable returns its current value and replaces it with the argument of the operation.

A $\text{fetch\_and\_add}$ operation returns the current value of a shared variable and adds its argument to the value of the variable. $\text{atomic\_add}$ is the same as $\text{fetch\_and\_add}$ except it just returns an acknowledgment that the operation is done, instead of the original value of the variable. $\text{fetch\_and\_inc}$ is a special case of $\text{fetch\_and\_add}$, which always adds 1 to the value of the shared variable.

In a distributed system, processes can be modeled as state machines. Each process has its own program counter to keep track of its location in its program code. The $\text{state}$ of a process consists of the values of all private variables owned by the process including the program counter. The $\text{configuration}$ of a system consists of the states of all processes and the values of all shared variables.

An $\text{event}$ consists of an atomic operation performed on a selected shared variable by a process, which may then update its state depending on its current state and the response of the operation. The value of the shared variable may also be updated. Thus, an event may change the system configuration. An $\text{execution fragment}$ of an algorithm is an alternating sequence of system configurations and events by processes, starting and ending with a configuration, where each configuration after the first results from the application of the preceding event to the preceding configuration. Note that an execution
\textbf{TEST\_AND\_SET} (V) \{ 
    \textbf{if} V = 0 \textbf{then} V := 1 
    \textbf{return} \text{TRUE} 
    \textbf{else return} \text{FALSE} \}

\textbf{COMPARE\_AND\_SWAP} (V, x, y) \{ 
    \textbf{if} V = x \textbf{then} V := y 
    \textbf{return} \text{TRUE} 
    \textbf{else return} \text{FALSE} \}

\textbf{LL} (V) \{ 
    X_V[p] := \text{TRUE} 
    \text{return} V \}

\textbf{SC} (V, x) \{ 
    \textbf{if} \neg X_V[p] \textbf{then return} \text{FALSE} 
    \textbf{for} i = 0 \textbf{to} N - 1 \textbf{do} X_V[i] := \text{FALSE} 
    V := x 
    \text{return} \text{TRUE} \}

\textbf{FETCH\_AND\_STORE} (V, x) \{ 
    t := V 
    V := x 
    \text{return} t \}

\textbf{FETCH\_AND\_ADD} (V, x) \{ 
    t := V 
    V := V + x 
    \text{return} t \}

\textbf{ATOMIC\_ADD} (V, x) \{ 
    V := V + x 
    \text{return} \text{ack} \}

\textbf{FETCH\_AND\_INC} (V) \{ 
    t := V 
    V := V + 1 
    \text{return} t \}

Figure 2.1: Pseudo-code for atomic operations
fragment can be infinite. An *execution* is an execution fragment that begins with an initial configuration, where all shared variables have their initial values, as specified by the algorithm. A configuration $C'$ is reachable from configuration $C$ in an algorithm, if there exists an execution fragment of the algorithm that starts with $C$ and ends with $C'$.

### 2.2 System models

In this thesis, two shared memory models are considered: the *distributed shared memory (DSM)* model [LH89] and the *cache-coherent (CC)* model [Tan76]. In the last two decades, many mutual exclusion algorithms were based on these models [AKH03]. Note that these models are not real systems, but abstract models. The description of these models lack of many details that real systems may have. There are also real systems based on variants of these models.

In the DSM model, shared memory is partitioned among all processes. Each process has its own local shared memory, and every shared variable is located in one of the local shared memories. A *local memory access* occurs when a process accesses a shared variable located in its own local memory. A *remote memory access (RMA)* occurs when a process accesses a shared variable located in another process’s local memory. CRAY T3E [Sco96] and MIT Alewife [ABC+95] are examples of systems that use the DSM model.

In the CC model, shared memory is remote from all processes, but each process has its own local cache. A shared variable is initially stored in the memory. When a process accesses a shared variable for the first time, it copies it into its local cache. A later read of the same shared variable by the process is a local memory access as long as, in the intervening time, no other process has performed an operation on this shared variable that might change its value. In the CC model, a shared variable can be copied to different caches, so it can be local to many processes at the same time.
In this model, the size of a cache is considered infinite, so a cached copy of a variable is not removed or replaced by another variable, which might occur in real systems due to the limitation of a cache size. Since we use only a small number of variables during busy-waiting, it is reasonable to assume that they will stay in the cache while processes are waiting.

The CC model requires a protocol for cache-consistency. In the CC model, an RMA occurs when a process accesses a variable for the first time, when the protocol updates a variable in main memory, when it invalidates caches, or when it copies the updated value of a variable from main memory to caches (or from a cache to other caches). The number of RMAs that occur during an execution and the amount of information carried by one RMA depend on the protocol. The information transferred through the network when updating a variable in main memory or caches is generally greater than the information transferred through the network when caches are invalidated.

If a system has a shared bus, it can be used to update or invalidate cached copies of a variable at the same time. Thus, in such systems, the cost for updating or invalidating cached copies of a variable does not depend on how many cached copies are updated or invalidated. However, in other systems, events that generate an RMA can cause different amounts of network traffic. For simplicity, papers that study local-spin algorithms only count the number of events that generate an RMA [GT90, MCS91, Cra93, YA95, AK01, GHHW07, AHW08]. In this thesis, we do the same thing.

There are three main protocols for maintaining cache consistency: the cache-update protocol [AB86], the cache-invalidate protocol with write-through [AP77], and the cache-invalidate protocol with write-back [Smi82]. When a process writes a new value to a shared variable, the cache-update protocol updates the value of this variable in main memory and all cached copies of this variable with the new value. Thus, all writes are remote memory accesses. In a system using this protocol, all read events, except the first read of a shared variable by a process, are local accesses. DEC Firefly [AB86] and Xerox
Dragon [AM87] are examples of systems that use the cache-update protocol.

In contrast, when a process writes a new value to a shared variable, the \textit{cache-invalidate protocol} only invalidates each cached copy, instead of updating it. In this protocol, different policies are used to determine when to update a variable in main memory. There are two main policies: write-through and write-back.

When a process writes to a variable, the \textit{cache-invalidate protocol with write-through} writes the new value to the variable in main memory, and it invalidates all cached copies of the variable at other processes. Hence, every write is a remote memory access. A read of variable $V$ by process $p$ retrieves the updated value from main memory, generating an RMA, if and only if $p$ has no valid cached copy of $V$. Subsequent reads of $V$ by $p$ do not generate RMAs until another process writes to $V$, invalidating $p$’s cached copy of $V$. An example of a system that uses only a cache-invalidate protocol with write-through is the Sequent Balance [TGF88].

In the \textit{cache-invalidate protocol with write-back}, when a new value is written to a variable, only the local cached copy of that variable is updated. It also invalidates all valid cached copies of the variable. A write to variable $V$ by process $p$ is an RMA if this is the first access to $V$ by $p$ or another process has a valid cached copy of $V$. As in the cache-invalidate protocol with write-through, a read of $V$ by process $q$ generates an RMA if and only if $q$ does not have a valid cached copy of $V$. If a cache miss occurs when $q$ wants to read $V$ and this read is the first access to $V$ since it was last updated by another process $p$, then the updated value is written to $V$ in main memory. Synapse [FI84] and Illinois [PP84] are early examples of the systems using cache-invalidate protocol with write-back.

The main advantage of the cache-update protocol is that it minimizes the number of cache misses by reads. Hence, an algorithm with frequent reads may benefit from using this protocol. Also, it has no overhead for recording which variables in a cache are invalid, unlike the cache-invalidate protocols. However, during a write, all cached
copies are updated, so the information transferred through the network by a write is
greater in the cache-update protocol than in cache-invalidate protocols. Moreover, the
cache update protocol does unnecessary updates when updated copies in caches are not
subsequently read. Thus, this protocol is often inefficient, and few systems support this
protocol.

The cache-invalidate protocol with write-through does an unnecessary update of main
memory if the updated value is not read by another process before the next write to the
same variable. In contrast, the cache-invalidate protocol with write-back updates a shared
variable in main memory only if another process reads the variable before the next update
of the variable. Hence, the cache-invalidate protocol with write-back is generally more
efficient than the cache-invalidate protocol with write-through. Another advantage of
the cache-invalidate protocol with write-back is that the information transferred through
the network by a write is small, since a write does not update main memory. Thus, an
algorithm with frequent writes may benefit from using this protocol. The cache-invalidate
protocol with write-back keeps track of which process has the newest value of a variable
in its cache, since a subsequent read copies the newest value to main memory. This is
unnecessary in the cache-invalidate protocol with write-through since the newest value is
always stored in main memory. Hence, write-back requires more overhead and is usually
more complex than write-through. Some multi-process systems such as DASH [LLG+92]
and SUN Niagara [KAO05] support multiple levels of cache, with write-through and
write-back used at different levels.

Now we compare these protocols in terms of the number of RMAs. In the cache-
update protocol, only the first read of a variable by a process generates an RMA, but
in the cache-invalidate protocol with write-back, a read also generates an RMA if the
process does not have a valid cached copy of the variable. Hence, the number of RMAs
generated by reads in the cache-invalidate protocol with write-back is at least as large as
in the cache-update protocol. All writes generate RMAs in the cache-update protocol,
but some writes do not generate RMAs in the cache-invalidate protocol with write-back. Hence, the number of RMAs generated by writes in the cache-invalidate protocol with write-back is smaller than or equal to the number of RMAs generated by the cache-update protocol. Hence, these two protocols are incomparable in terms of the number of RMAs they generate.

In both the cache-update protocol and the cache-invalidate protocol with write-through, all writes generate RMAs. In the cache-invalidate protocol with write-through, a read of a variable generates an RMA if the process does not have a valid cached copy of the variable. However, in the cache-update protocol, only the first read of a variable generates an RMA. Hence, the number of RMAs generated by reads in the cache-invalidate protocol with write-through is at least as large as in the cache-update protocol. Therefore, the total number of RMAs in a system using the cache-invalidate protocol with write-through is always at least as great as (and is usually greater than) in a system using the cache-update protocol.

Both cache-invalidate protocols generate one RMA for each read of a variable by a process that does not have a valid cached copy of the variable. The cache-invalidate protocol with write-through generates one RMA for each write, but the cache-invalidate protocol with write-back does not generate an RMA for some writes. Hence, the number of RMAs generated using write-through is greater than or equal to the number of RMAs generated using write-back. Therefore, the number of RMAs used by the cache-invalidate protocol with write-through is an upper bound on the number of RMAs used in the other two variants of the CC model.

2.3 The mutual exclusion problem and its variants

In the mutual exclusion problem, each process performs a sequence of passages, such that, during each passage, it enters and exits the critical section. Formally, a passage is
the sequence of steps performed by a process from when it begins the trying protocol until it next returns to the remainder section. We say that a process is active if and only if it is not in the remainder section. Processes executing the trying protocol are called trying processes. Process \( p \) is in its \( i \)th passage at some point in an execution if and only if \( p \) is active and \( p \) has invoked the trying protocol \( i \) times (including its current passage) prior to this point. Note that, since we assume there is no process failure, each active process eventually performs its next step.

Any mutual exclusion algorithm must satisfy at least two properties: mutual exclusion and deadlock freedom. The mutual exclusion property prevents more than one processes from being in the critical section simultaneously, and the deadlock freedom property ensures that the critical section is always available to some process.

(R1) **Mutual exclusion:** No two processes are in the critical section at the same time.

(R2) **Deadlock freedom:** If there is a process in the trying protocol, then some process eventually enters the critical section. If there is a process in the exit protocol, then some process eventually returns to the remainder section.

Although a mutual exclusion algorithm satisfies deadlock freedom, some process in the trying protocol might never enter the critical section. Hence, lockout freedom, which is stronger than deadlock freedom, is often required.

(R2') **Lockout freedom:** A process in the trying protocol eventually enters the critical section. A process in the exit protocol eventually returns to the remainder section.

A waiting period in the trying protocol is inevitable in the mutual exclusion problem, since a process must wait when the critical section is occupied by another process. However, a waiting period in the exit protocol is not necessary and it is more desirable for a process to return to the remainder section without waiting.

(R3) **Bounded exit:** A process in the exit protocol finishes the protocol within a bounded number of its own steps.
Throughout this thesis, we say that an algorithm solves mutual exclusion if it satisfies (R1), (R2'), and (R3).

In real asynchronous systems, there are various factors that affect the rate at which processes take steps, including preemption, scheduling, the number of virtual processes simulated by one processor, and memory access latency. Thus, even in a multi-processor system in which all processors have the same clock speed, some processes may take steps more slowly than others. In a mutual exclusion algorithm, a slow process can be delayed by fast processes, even if the fast processes entered the trying protocol after the slow process. Hence, a mutual exclusion algorithm can be unfair to slow processes.

One way to avoid this problem is to restrict the number of times each process waits for other processes in the trying protocol. For example, we might want to let passages enter the critical section in the order they started. However, when a process begins a passage, other processes cannot learn about it until it has changed the value of a shared variable. A *doorway* is a section of code at the beginning of the trying protocol in which a process performs a bounded number of steps. In a queue-based algorithm, passages enter the critical section in the same order as they finish the doorway. Note that each process can invoke a mutual exclusion algorithm many times, so passages, not processes, are ordered.

(R5) **Queue-based**: If a passage $I$ finishes the doorway before another passage $J$ finishes the doorway, then $I$ enters the critical section before $J$ does.

Some algorithms do not order passages according to the times they finish the doorway. Instead, they use a partial order, which gives a weaker version of (R5) called *first-come first-served (FCFS)*.

(R6) **FCFS**: If a passage $I$ finishes the doorway before another passage $J$ starts (the doorway), then $I$ enters the critical section before $J$ does.

Note that, if a passage $I$ finishes the doorway after another passage $J$ starts the doorway
and before $J$ finishes the doorway, then, in queue-based algorithms, $I$ enters the critical section before $J$ does, but, in FCFS algorithms, either $I$ or $J$ can enter the critical section first. Also note that, if an algorithm satisfies both the FCFS and deadlock freedom properties, then it also satisfies the lockout freedom property.

The *abortable mutual exclusion problem* is the same as the mutual exclusion problem, except a process that is waiting in the trying protocol can *abort*, i.e., it can stop waiting and return to the remainder section (without entering the critical section or exit protocol). When a process wants to abort, it performs a bounded section of code, called an *abort protocol*. Note that, in the trying protocol, a process performs a bounded number of its own steps before entering a waiting period. Hence, if a process wants to abort in the trying protocol when it is not waiting, it continues performing the trying protocol until it reaches a waiting period and then performs the abort protocol. An abortable mutual exclusion algorithm must satisfy the following requirements in addition to (R1) and (R3).

(R2″) **Lockout freedom**: If a process is in the trying protocol and does not abort, then it eventually enters the critical section. A process in the exit protocol eventually returns to the remainder section.

(R7) **Bounded abort**: A process in the abort protocol finishes the protocol within a bounded number of its own steps.

Requirement (R2″) is the same as (R2′) except that it only applies to a process that has not aborted since it last began the trying protocol. Also note that requirement (R7) is a crucial requirement of the abortable mutual exclusion problem: A process that wants to abort because it has waited too long gets no benefit from an abort protocol that involves waiting.

A variant of (R5) or (R6) may be added to these requirements.

(R5′) **Queue-based**: If a passage $I$ finishes the doorway before another passage $J$ finishes the doorway and $I$ does not abort, then $I$ enters the critical section before
(R6’) **FCFS**: If a passage $I$ finishes the doorway before another passage $J$ starts the doorway and $I$ does not abort, then $I$ enters the critical section before $J$ does.

In the $k$—**exclusion problem** [FLBB79], which is a generalization of the mutual exclusion problem, the shared resource can be accessed by at most $k$ processes at the same time. A $k$—exclusion algorithm must satisfy the following requirements in addition to (R2’) and (R3).

(R1’) **$k$—exclusion**: no more than $k$ processes are in the critical section at the same time.

(R8) **$k$—deadlock freedom**: If there is a process in the trying protocol and there are at most $k - 1$ processes in the critical section, then some process in the trying protocol eventually enters the critical section within a bounded number of its own steps.

Note that 1—exclusion is another name for mutual exclusion. Requirement (R8) ensures that the critical section is not underutilized.

A crash failure occurs when a process stops taking steps while it is in the trying protocol, critical section, or exit protocol. Formally, process $p$ has a special event, $stop_p$, which can be performed at any time and after which $p$ does not perform any events. Process $p$ is crashed in $C$ if $p$ is not in the remainder section in $C$ and $stop_p$ was performed before $C$. Event $stop_p$ does not change any shared variables, so other processes cannot distinguish between a process that has crashed and a process that is very slow. If a process is crashed while it is in the critical section, no other process can subsequently enter the critical section, so deadlock will occur. In this thesis, we assume that there are no crash failures.

Likewise, in $k$—exclusion, if $k$ or more processes crash in the critical section, deadlock will occur. If a process crashes in the trying protocol, lockout freedom and $k$—deadlock
freedom are violated. Thus, in a model that allows crash failures, (R2′) and (R8) only apply to correct processes and only if at most \( k - 1 \) processes crashed. Also (R3) only applies to correct processes. [FLBB79, AM97]

Another generalization of the mutual exclusion problem is *group mutual exclusion*, introduced by Joung [Jou98]. Suppose we have many shared resources, but only one resource is available at a time. In this problem, when a process enters the trying protocol, it requests a specific resource. Processes requesting the same resource can be in the critical section at the same time, but processes requesting different resources may not be in the critical section at the same time. More precisely, group mutual exclusion algorithms must satisfy the following property in addition to (R2′) and (R3).

(R1′′) **Group mutual exclusion**: If two passages are in the critical section at the same time, then they requested the same resource.

Note that (R1) implies (R1′′). Hence, to ensure that the processes that want the same resource are allowed to enter the critical section without waiting, group mutual exclusion algorithms must satisfy the following property in addition to (R1′′), (R2′) and (R3).

(R9) **Concurrent entering**: If a passage \( I \) requests a resource and no other passage in the trying protocol or critical section requests a different resource, then \( I \) enters the critical section within a bounded number of its own steps.

Mutual exclusion is a special case of group mutual exclusion where each process requests a different resource: for example, process \( i \) always requests resource \( R_i \).

### 2.4 Local-spin algorithms and RMA complexity

The RMA complexity of an algorithm is the worst case number of RMAs performed by a process during one of its passages. Note that the number of passages is unbounded, since each process may invoke the trying protocol an unbounded number of times.
In mutual exclusion algorithms, a trying process keeps accessing shared variables when it is waiting for the critical section. This period is called a busy-waiting period. Formally, process \( p \) is \textit{busy-waiting} in configuration \( C \) if \( p \) is in the trying protocol in \( C \) and there exists an infinite solo execution fragment \( E_p \) performed by \( p \) starting at \( C \), such that \( p \) is in the trying protocol throughout \( E_p \) and there are infinitely many configurations \( C' \) in \( E_p \) where the program counter of \( p \) in \( C' \) is the same as the program counter of \( p \) in \( C \). The shared variables that are accessed infinitely many times by process \( p \) during \( E_p \) are called \textit{spin variables}. Note that there must be at least one spin variable, unless \( p \) accesses infinitely many different shared variables during \( E_p \). A \textit{busy-waiting period} of process \( p \) in some execution is a maximal fragment of the execution such that \( p \) is busy-waiting in all of its configurations.

Since there is no bound on the number of times a process accesses its spin variables during a busy-waiting period, accesses to the spin variables must be local to the waiting process in order for the algorithm to have bounded RMA complexity. We say that an algorithm is \textit{local-spin} if there exists a bound \( B \) such that, for all busy-waiting periods in any execution of the algorithm, each process performs at most \( B \) RMAs.

The local-spin property puts different constraints on an algorithm depending on the model. In the DSM model, each spin variable must be local to the process that waits for it. During busy-waiting, a process only reads its spin variables in the CC model using a cache-update protocol. In the CC model using a cache-invalidate protocol with write-through, spin variables can be updated only a bounded number of times during a busy-waiting period, since each update causes a cache miss for the waiting process. In the CC model using a cache-invalidate protocol with write-back, a process can update a variable many times during a busy-waiting period provided there is a bound on the number of times other processes access that variable during this period.
Chapter 3

Related Work

T. Dekker developed the first 2-process algorithm that satisfies mutual exclusion and deadlock freedom [Dij68]. In 1965, Dijkstra [Dij65] presented the first N-process algorithm. The first mutual exclusion algorithm that prevents lockout was presented by Knuth [Knu66] in 1966. These classic algorithms use only registers, but they are not local-spin.

Lamport’s bakery algorithm was the first mutual exclusion algorithm that satisfies the FCFS property [Lam74]. Lamport’s algorithm is local-spin in the CC model, but its RMA complexity is \( \Theta(N) \). It works even with certain kinds of non-atomic registers, but they have unbounded size.

Peterson and Fischer [PF77] presented a reduction from N-process mutual exclusion to 2-process mutual exclusion, using a tournament tree of height \( \Theta(\log N) \). In the tournament tree, each internal node is a 2-process mutual exclusion algorithm and each process is assigned to a different leaf. When a process enters the critical section of the 2-process mutual exclusion algorithm at an internal node, it participates in the 2-process mutual exclusion algorithm at its parent node. If a process enters the critical section at the root, then it enters the critical section of the whole algorithm. When a process finishes the critical section, it performs the exit protocols of the 2-process algorithms in which it
participated, in reverse order.

Anderson et al. [AKH03] presented a survey of mutual exclusion algorithms for shared memory models, including local-spin mutual exclusion and mutual exclusion using non-atomic registers.

In the rest of this chapter, I discuss some local-spin mutual exclusion algorithms that are related to my abortable mutual exclusion algorithms. I also survey existing local-spin abortable mutual exclusion algorithms and \( k \)-exclusion algorithms. Lower bounds on the RMA complexity of the mutual exclusion problem are summarized. Finally, I describe existing memory reclamation methods.

### 3.1 Local-spin mutual exclusion algorithms

In Lamport’s bakery algorithm, each process performs \( \Theta(N) \) RMAs to enter the critical section in the CC model. Code for this algorithm is presented in Figure 1. In this algorithm, process \( i \) starts its doorway by setting \( \text{Doorway}[i] \) to \text{TRUE}. Then it chooses a ticket that is larger than the ticket presently held by any other process. It completes the doorway by resetting \( \text{Doorway}[i] \) to \text{FALSE}. Then it waits for its ticket to be the smallest among the tickets of all trying processes. In the trying protocol, each process \( p \) waits for every other process twice. The first time process \( p \) waits for process \( i \), it waits for \( \text{Doorway}[i] \) to become \text{FALSE}. Hence, this waiting period ends when process \( i \) is outside the doorway. The second time process \( p \) waits for process \( i \), it waits for \( \text{Ticket}[i] \) to become either zero or a value greater than \( p \)'s ticket. \( \text{Ticket}[i] \) holds process \( i \)'s ticket and becomes zero when \( i \) is in the remainder section. Hence, this waiting period ends when process \( i \) either is in the remainder section or has received a ticket greater than \( p \)'s. In total, each process has at most \( 2N - 2 \) waiting periods prior to entering the critical section. Because the algorithm satisfies the FCFS property, each of \( \text{Doorway}[i] \) and \( \text{Ticket}[i] \) is updated at most twice during one passage by \( p \). Hence, each waiting
Figure 1 Bakery Algorithm (Code for process $p$)

shared variables:
- Ticket: array\[1..N\] of N, initially 0
- Doorway: array\[1..N\] of Boolean, initially false

private variable:
- $t, \text{max} \in \mathbb{N}$
- $i \in \{1, \ldots, N\}$

TryingProtocol()

1: Doorway\[p\] := true
2: for $i = 1$ to $N$ do
3: \hspace{1em} $t := \text{Ticket}[i]$
4: \hspace{1em} if max $< t$ then
5: \hspace{1em} \hspace{1em} max := $t$
6: end if
7: end for
8: Ticket\[p\] := $1 + \text{max}$
9: Doorway\[p\] := false  
   \hspace{1em} \% Doorway ends
10: for $i = 1$ to $N$ do
11: \hspace{1em} await $\neg \text{Doorway}[i]$
12: \hspace{1em} $t := \text{Ticket}[i]$
13: \hspace{1em} while $t \neq 0 \land (t, i) < (1 + \text{max}, p)$ do
14: \hspace{1em} \hspace{1em} $t := \text{Ticket}[i]$
15: \hspace{1em} end while
16: end for

ExitProtocol()

17: Ticket\[p\] := 0

period takes $O(1)$ RMAs in the CC model, and the RMA complexity of the algorithm is $\Theta(N)$.

In Lamport’s bakery algorithm, many processes spin on $\text{Doorway}[i]$ and $\text{Ticket}[i]$. Hence the algorithm is not local spin in the DSM model. Taubenfeld [Tau04] modified Lamport’s bakery algorithm so that the resulting algorithm is local-spin in the DSM model, by using $2N^2$ Boolean spin variables, $\text{SpinD}[p, q]$ and $\text{SpinT}[p, q]$, for all processes $p$ and $q$. Both $\text{SpinD}[p, q]$ and $\text{SpinT}[p, q]$ are local to process $p$. $\text{SpinD}[p, q]$ and $\text{SpinT}[p, q]$ are set to true only by process $p$ and set to false only by process $q$. At the end of the doorway, process $i$ sets $\text{Doorway}[i]$ to false, and then sets $\text{SpinD}[p, i]$ to false for every process $p$. Instead of waiting for $\text{Doorway}[i]$ to become false, as
Figure 2 Bakery Algorithm for the DSM model (Code for process $p$)

shared variables:

- $Ticket$: array[1..N] of N, initially 0
- $Doorway$: array[1..N] of Boolean, initially FALSE
- $SpinD$, $SpinT$: array[1..N][1..N] of Boolean, initially TRUE

/* For $i = 1, \ldots, N$, $SpinD[p, i]$ and $SpinT[p, i]$ are in $p$’s memory */

private variable:

$t, max \in \mathbb{N}, i \in \{1, \ldots, N\}$

TryingProtocol()

1: $Doorway[p] := TRUE$
2: for $i = 1$ to $N$ do
3: $t := Ticket[i]$
4: if $max < t$ then
5: $max := t$
end if
end for
6: $Ticket[p] := 1 + max$
7: $Doorway[p] := FALSE$
% Doorway ends
8: for $i = 1$ to $N$ do
9: $SpinD[i, p] := FALSE$
end for
10: for $i = 1$ to $N$ do
11: $SpinD[p, i] := TRUE$
12: if $Doorway[i]$ then
13: await $\neg SpinD[p, i]$
end if
14: $SpinT[p, i] := TRUE$
15: $t := Ticket[i]$
16: if $t \neq 0 \land (t, i) < (1 + max, p)$ then
17: await $\neg SpinT[p, i]$
end if
end for

ExitProtocol()

18: $Ticket[p] := 0$
19: for $i = 1$ to $N$ do
20: $SpinT[i, p] := FALSE$
end for

in the original bakery algorithm, process $p$ sets $SpinD[p, i]$ to TRUE, reads $Doorway[i]$, and, if it is TRUE, waits for $SpinD[p, i]$ to become FALSE. In this way, process $p$ accesses $Doorway[i]$ only once and spins on a local variable. The second time process $p$ waits for process $i$ is modified similarly.

This modification forces each process to write to $\Theta(N)$ remote memory locations.
However, since the bakery algorithm has $\Theta(N)$ RMA complexity in the CC model, these additional writes do not affect the overall RMA complexity. Hence, the RMA complexity of the resulting algorithm is $\Theta(N)$ in the DSM model. Figure 2 shows this algorithm in detail, where the modifications are shaded.

Both Lamport’s bakery algorithm and its modification by Taubenfeld use unbounded space because ticket numbers grow without bound. Taubenfeld [Tau04] also describes a variant with bounded space. The algorithm uses tickets of two colors, white and black. When a process gets a ticket in the trying protocol, its color is different than the ticket held by the last process to finish the critical section, or the color white, if no process has yet finished the critical section. The value of its ticket is one greater than the maximum of the values of all tickets with the same color. If there are no other tickets with the same color, the value of its ticket is one. Processes whose tickets do not have the same color as the ticket of the last process to finish the critical section have lower priority than those whose do.

Suppose a process with a white ticket finishes the critical section. Then, all processes with white tickets enter the critical section before any process with a black ticket. Until a process with a black ticket subsequently finishes the critical section, only black tickets are dispensed. Since there are $N$ processes, at most $N$ white tickets are dispensed consecutively. When a process finishes the critical section, the value of its ticket becomes zero. Hence, the first time a process with a black ticket subsequently finishes the critical section, there is no process with a white ticket. When processes in the doorway subsequently receive white tickets, their values start at one. Therefore, the value of every ticket is bounded above by $N$.

There are two other approaches used by local-spin mutual exclusion algorithms: One is based on a $\Theta(\log N)$-height tournament tree, as in Peterson and Fischer’s transformation [PF77], and the other is based on a queue.

Tree-based algorithms have $\Theta(\log N)$ RMA complexity. Most of them use only reg-
isters and do not have the FCFS property. However, Danek and Golab [DG08] gave a transformation from any mutual exclusion algorithm to a FCFS mutual exclusion algorithm using only registers, which incurs an additional $O(\log N)$ RMAs.

In queue-based algorithms, trying processes form a queue and each waiting process waits only for its predecessor in the queue. Processes are allowed to enter the critical section in the order they are enqueued. All known queue-based mutual exclusion algorithms have $O(1)$ RMA complexity and satisfy the FCFS property. They all use powerful atomic operations like `fetch_and_store` or `fetch_and_inc`.

### 3.1.1 Tree-based local-spin mutual exclusion algorithms

Tree-based local-spin mutual exclusion algorithms have the same structure as Peterson and Fischer’s tournament tree algorithm [PF77]. An instance of a 2-process local-spin mutual exclusion algorithm is located at each internal node of the tree, and each process is located at a different leaf of the tree. Although at most two processes can simultaneously participate in the 2-process algorithm at some node, any process located at a leaf of the subtree rooted at the node can perform the 2-process algorithm. Hence, the number of process ID’s that can participate in this instance of the algorithm is the same as the number of leaves in the subtree rooted at the node.

Consider a tree-based $N$-process mutual exclusion algorithm. Suppose that, at each internal node, it uses a 2-process mutual exclusion algorithm that is local-spin in a CC model and satisfies the FCFS property. Then the $N$-process algorithm is also local-spin in this CC model. To see why, suppose process $p$ is busy-waiting at some node in the $N$-process mutual exclusion algorithm and some other process $q$ is also participating in the 2-process algorithm at this node. Since the 2-process mutual exclusion algorithm is local-spin in the CC model, there is an upper bound on the number of times $q$ accesses $p$’s spin variables. If $q$ finishes the exit protocol of the 2-process algorithm and another process $r$ invokes the 2-process algorithm of this node, then the FCFS property implies
that \( p \) enters the critical section of the 2-process algorithm before \( r \) does. Since the 2-process mutual exclusion algorithm is local-spin, \( r \) accesses \( p \)'s spin variables a bounded number of times while \( p \) is busy-waiting. Therefore, while \( p \) is busy-waiting at some node in the \( N \)-process algorithm, \( p \)'s spin variables are accessed a bounded number of times by other processes. Hence, the \( N \)-process algorithm is also local-spin in this CC model.

In the DSM model, the same argument does not hold, since each variable is only local to one process. Although only two processes can participate in the 2-process algorithm at a particular node at the same time, there may be a larger number, say \( M \), that can participate at different times. Since each such process must spin on a local variable and each variable is only local to one process in the DSM model, the algorithm at this node must use at least \( M \) different spin variables. Thus, tree-based mutual exclusion algorithms that are local-spin in the DSM model and use different spin variables at each level of the tree use a total of \( \Omega(N \log N) \) spin variables.

Also, in each 2-process algorithm, if process \( p \) is in the critical section and another process, \( q \), is busy-waiting in the trying protocol, then \( p \) must learn \( q \)'s ID so that, in its exit protocol, \( p \) can change the value of a local variable on which \( q \) is spinning. If \( p \) does not access \( q \)'s spin variables in the exit protocol, then \( q \) might wait forever, since it is possible that no process subsequently invokes the 2-process algorithm. Hence, before \( q \) waits in the trying protocol, \( q \) must write its ID to a shared variable that \( p \) reads in its exit protocol.

Yang and Anderson [YA95] developed the first 2-process mutual exclusion algorithm that is local-spin in the DSM model. In this algorithm, which appears in Figure 3, each process is given a side, either 0 or 1, according to its ID, and two processes on the same side cannot be active at the same time. The algorithm has a doorway at the beginning of the trying protocol, and it allows the process that finishes the doorway first to enter the critical section first. To determine whether a process from the other side has finished the doorway, the algorithm has an additional waiting period before the essential waiting
Figure 3 Yang and Anderson’s two process algorithm (Code for process $p$)

**shared variables:** ($L$ is the set of process IDs that can participate in this algorithm.)
- **Want:** array[0,1] of $L \cup \{\text{NIL}\}$, initially NIL
- **LastIn** $\in L$
- **Spin:** array of \{RED, YELLOW, GREEN\} indexed by $L$, initially RED
  ($\text{Spin}[p]$ is located in $p$’s local memory.)

**private constant:**
- $side = 0$ if $p$ is in the left subtree of the node; otherwise, $side = 1$

**private variable:**
- **rival** $\in L \cup \{\text{NIL}\}$

**TryingProtocol()**
1. $\text{Want}[side] := p$
2. $\text{LastIn} := p$
   \%
3. $\text{Spin}[p] := \text{RED}$
4. $\text{rival} := \text{Want}[1 - side]$
5. if $\text{rival} \neq \text{NIL}$ then
6.   if $\text{LastIn} = p$ then
7.     if $\text{Spin}[\text{rival}] = \text{RED}$ then
8.       $\text{Spin}[\text{rival}] := \text{YELLOW}$
     \end if
9.   end if
10.   await $\text{Spin}[p] \neq \text{RED}$
11. end if
12. if $\text{LastIn} = p$ then
13.   await $\text{Spin}[p] = \text{GREEN}$
14. end if
15. end if

**ExitProtocol()**
1. $\text{Want}[side] := \text{NIL}$
2. $\text{rival} := \text{LastIn}$
3. if $\text{rival} \neq p$ then
4.   $\text{Spin}[\text{rival}] := \text{GREEN}$
   \end if

period in which a process waits for the other process to finish the critical section.

A more detailed description follows. In the doorway, a trying protocol on side $s$ announces its invocation by writing its ID to the shared variable $\text{Want}[s]$, and then writes its ID to the shared variable $\text{LastIn}$. The initial values of both components of $\text{Want}$ are NIL. $\text{Want}[s]$ is set back to NIL when a process on side $s$ begins its exit protocol. Hence, the value of $\text{Want}[s]$ contains the ID of the process on side $s$ that is currently in the trying protocol or critical section. The value of $\text{LastIn}$ contains the ID
of the process that last finished the doorway.

After finishing line 3, a process \( p \) on side \( s \) checks the values of \( \text{Want}[1-s] \) and \( \text{LastIn} \) to determine whether it can enter the critical section without waiting. If \( \text{Want}[1-s] = \text{NIL} \), then no process on the other side is in the critical section and \( p \) enters the critical section immediately. If \( \text{Want}[1-s] \neq \text{NIL} \) and \( \text{LastIn} \neq p \), then there is a process on the other side, and \( p \) finished the doorway before that other process. In this case, \( p \) also enters the critical section immediately, and the other process will wait for \( p \).

If \( \text{Want}[1-s] = q \neq \text{NIL} \) and \( \text{LastIn} = p \), then there are two cases: either \( p \) finished the doorway after the other process, \( q \), or \( p \) finished the doorway before \( q \), but \( q \) has not yet finished the doorway. Process \( p \) can distinguish between these two cases after \( q \) finishes its doorway. Each process \( p \) uses one three-valued spin variable, \( \text{Spin}[p] \), which it reads from. It writes to \( \text{Spin}[q] \). \( \text{Spin}[p] \) has value RED, YELLOW, or GREEN.

When \( p \) is between line 6 and line 11, RED indicates process \( p \) cannot terminate either busy-waiting period; YELLOW indicates \( p \) can terminate the first busy-waiting period, but not the second; and GREEN indicates \( p \) can terminate both busy-waiting periods. Before \( p \) begins waiting (on line 9), it changes \( \text{Spin}[q] \) from RED to YELLOW (on line 8). Then, it waits until \( \text{Spin}[p] \) is no longer RED. If \( \text{Spin}[p] \) becomes GREEN, then \( p \) can enter the critical section. If \( \text{Spin}[p] \) becomes YELLOW, \( q \) has finished the doorway, so the value of \( \text{LastIn} \) indicates which of the two processes finished the doorway last. If \( \text{LastIn} = p \), then \( p \) waits for \( q \) to finish the critical section; otherwise, \( p \) enters the critical section immediately.

In the exit protocol, \( p \) checks \( \text{LastIn} \) again after setting \( \text{Want}[s] \) to NIL. If \( \text{LastIn} = q \neq p \), then the other process, \( q \), finished the doorway after \( p \) did. In this case, \( p \) changes \( \text{Spin}[q] \) to GREEN, to allow \( q \) to enter the critical section.

The RMA complexity of Yang and Anderson’s \( N \)-process algorithm is \( \Theta(\log N) \), because each mutual exclusion algorithm at an internal node has \( O(1) \) RMA complexity. This follows from the facts that Yang and Anderson’s 2-process algorithm has \( O(1) \) RMA
complexity in the DSM model, and $Spin[p]$ is updated by at most three times during one passage by process $p$, including when $Spin[p]$ is set to RED by $p$ on line 3. Let’s see why this second fact is true. In its trying protocol, process $q$ writes to $Spin[p]$ on line 8, which is performed at most once in each passage. This write changes $Spin[p]$ from RED to YELLOW. It occurs only if process $p$ is in the doorway, or $p$ finished the doorway before $q$ did. In both cases, we now show that, during one passage by $p$, no matter how many times the trying protocol on the other side of the 2-process algorithm is invoked, line 8 is performed at most once.

In the first case, a process performing the trying protocol on the other side of the 2-process algorithm waits on line 9 for $p$ to finish its doorway and change its spin variable from RED to YELLOW. Then it enters the critical section. In its exit protocol, this process changes $Spin[p]$ to GREEN, since $p$ finished the doorway after it did and, hence, $LastIn = p$. Note that $p$ set $Spin[p]$ to RED in its trying protocol before this process entered the critical section. Hence, $Spin[p]$ is not set back to RED after it is set to GREEN unless $p$ returns to the remainder section and re-invokes the trying protocol. Thus, if the trying protocol on the other side is re-invoked before $p$ returns to the remainder section, it does not change $Spin[p]$ from RED to YELLOW on line 8.

In the second case, $p$ enters the critical section before a process performing the trying protocol on the other side of the 2-process algorithm. When $p$ finishes the critical section, it changes $Want[s]$ to NIL, where $s$ is the side on which $p$ is located. Hence, if the trying protocol on the other side is invoked by any process $q$ before $p$ returns to the remainder section, then $q$ enters the critical section without performing lines 6 to 11 and, hence, without changing $Spin[p]$.

Finally, we show that $Spin[p]$ is changed in the exit protocol on the other side of the 2-process algorithm at most once during one passage by $p$, no matter how many times the exit protocol on the other side is invoked. Suppose some process on the other side writes to $Spin[p]$ in some invocation of its exit protocol. Then, when this process last
read LastIn, it had value \( p \). Hence, process \( p \) was the last process to finish the doorway. If the trying protocol on the other side is later invoked by some process \( q \) during the same passage by \( p \), then LastIn is set to \( q \). Thus, \( q \) will not write to \( \text{Spin}[p] \) in later invocations of its exit protocol during the same passage by \( p \).

Yang and Anderson’s 2-process algorithm uses three shared variables, \( \text{Want}[0] \), \( \text{Want}[1] \), and LastIn. At the root of the tournament tree, it also uses \( N \) spin variables, \( \text{Spin}[1], \ldots, \text{Spin}[N] \), one for each process. However, if a 2-process algorithm is at the root of a subtree with \( M \) leaves, then it only uses \( M \) spin variables. In Yang and Anderson’s \( N \)-process algorithm, the tournament tree consists of \( N - 1 \) different instances of this 2-process algorithm, and each process participates in at most \( \lceil \log_2 N \rceil \) 2-process algorithms. Hence, it uses at most \( \lceil \log_2 N \rceil \) spin variables for each process, for a total of \( N \lceil \log_2 N \rceil \) spin variables. Each internal node in the tree uses a separate copy of the three other shared variables, for a total of \( 3(N - 1) \) shared variables. Hence, the space complexity of the \( N \)-process algorithm is \( \Theta(N \log N) \).

Yang and Anderson’s algorithm becomes incorrect if each process uses only one spin variable for the entire tournament tree. To see why, consider the following example. Suppose that process \( p \) begins the trying protocol at a node after entering the critical section at one of its children. Since \( p \) is in the critical section at this child, the other process \( q \) at this child can change \( \text{Spin}[p] \) from RED to YELLOW in its trying protocol. Suppose that, in the trying protocol at the parent node, \( p \) is waiting for \( \text{Spin}[p] \) to be changed from RED. This is supposed to happen when the other process \( r \) at the parent node has begun the trying protocol but has not finished the doorway. However, if \( \text{Spin}[p] \) is changed from RED to YELLOW by \( q \) at the child node, then \( p \) stops waiting at the parent node, and checks the value of LastIn. Since \( r \) has not finished the doorway, LastIn = \( p \), so \( p \) thinks that it finished the doorway after \( r \) and waits for \( r \) to finish the critical section. At the end of the doorway at the parent node, \( r \) sets LastIn to \( r \) and reads LastIn. Then \( r \) thinks that it finished the doorway after \( p \), so it waits for \( p \) to
finish the critical section. This causes a deadlock at the parent node.

Kim and Anderson [KA02] modified Yang and Anderson’s algorithm by adding two additional shared variables per node, so that each process uses only one spin variable for the entire tournament tree. The resulting algorithm uses only $\Theta(N)$ shared variables including spin variables, so its space complexity is $\Theta(N)$. Although a process may perform $\Theta(\log N)$ RMAs in a 2-process algorithm at an internal node, the overall RMA complexity of the algorithm is still $\Theta(\log N)$.

We describe Kim and Anderson’s algorithm in more detail. Figure 4 shows Kim and Anderson’s 2-process algorithm, where modifications are shaded. Each process has one spin variable, $Spin’[p]$. At each internal node of the tree, instead of using $Spin[p]$ and $Spin[q]$, they use two three-valued shared variables, $Check[0]$ and $Check[1]$, in addition to $Spin’[p]$ and $Spin’[q]$. Instead of waiting for $Spin[p]$ to be changed, process $p$ checks the value of $Check[s]$, where $s$ is the side on which $p$ is located. If $p$ has to wait, then $p$ uses the Boolean spin variable $Spin’[p]$ and waits for this variable to be changed to TRUE. When the other process, $q$, changes $p$’s spin variable, it changes $Check[s]$ and then $Spin’[p]$. When $p$ finishes waiting, it resets $Spin’[p]$ to FALSE and then checks the value of $Check[s]$ again. Since $Check[s]$ can be accessed only by the processes that participate in the 2-process algorithm at this internal node, $Check[s]$ is changed at the last step performed by $q$ prior to it changing $Spin’[p]$. If $Check[s]$ was changed, then $p$ finishes this waiting period. Otherwise, $Spin’[p]$ was changed by a process at a different node. In this case, $p$ waits for $Spin’[p]$ again and repeats the same test when it finishes waiting.

3.1.2 Queue-based local-spin mutual exclusion algorithms

T. Anderson [And90] developed the first local-spin queue-based mutual exclusion algorithm in the CC model. He used a circular list of $N$ Boolean variables for a queue of trying processes. All slots of the list except one are initially FALSE. Each process gets the position of a slot of the circular list using FETCH_AND_INC modulo $N$ on a variable that
Figure 4 Kim and Anderson’s two process algorithm (Code for process \( p \))

**shared variables:** (\( L \) is the set of process IDs that can participate in this algorithm.)
- \( \text{Want: array}[0,1] \) of \( L \cup \{\text{NIL}\} \), initially NIL
- \( \text{LastIn} \in L \).
- \( \text{Check: array}[0,1] \) of \{RED, YELLOW, GREEN\}, initially RED.
- \( \text{Spin'}: \) array of Boolean, indexed by \( L \), initially FALSE.
  (\( \text{Spin'}[p] \) is located in \( p \)’s local memory.)

**private constant:**
- \( \text{side} = 0 \) if \( p \) is in the left subtree of the node; otherwise, \( \text{side} = 1 \)

**private variable:**
- \( \text{rival} \in L \cup \{\text{NIL}\} \)

**TryingProtocol()**
1: \( \text{Want}[\text{side}] := p \)
2: \( \text{LastIn} := p \)
  \( \% \) Doorway ends
3: \( \text{Check}[\text{side}] := \text{RED} \)
4: \( \text{rival} := \text{Want}[1 – \text{side}] \)
5: if \( \text{rival} \neq \text{NIL} \) then
6: if \( \text{LastIn} = p \) then
7: if \( \text{Check}[1 – \text{side}] = \text{RED} \) then
8: \( \text{Check}[1 – \text{side}] := \text{YELLOW} \)
9: \( \text{Spin'}[\text{rival}] := \text{TRUE} \)
end if
10: while \( \text{Check}[\text{side}] = \text{RED} \) do
11: \( \text{await} \text{Spin'}[p] \)
12: \( \text{Spin'}[p] := \text{FALSE} \)
end while
13: if \( \text{LastIn} = p \) then
14: while \( \text{Check}[\text{side}] \neq \text{GREEN} \) do
15: \( \text{await} \text{Spin'}[p] \)
16: \( \text{Spin'}[p] := \text{FALSE} \)
end while
end if
end if

**ExitProtocol()**
1: \( \text{Want}[\text{side}] := \text{NIL} \)
2: \( \text{rival} := \text{LastIn} \)
3: if \( \text{rival} \neq p \) then
4: \( \text{Check}[1 – \text{side}] := \text{GREEN} \)
5: \( \text{Spin'}[\text{rival}] := \text{TRUE} \)
end if
points to the tail of the queue. A process waits for its slot to be TRUE before entering the critical section. When a process enters the critical section, it sets its slot back to FALSE. After finishing the critical section, a process sets the following slot to TRUE to allow its successor to enter the critical section. This algorithm has $O(1)$ RMA complexity in the CC model.

Graunke and Thakkar [GT90], Craig [Cra93], and Magnusson, Landin and Hagersten [MLH94] also developed local-spin queue-based mutual exclusion algorithms in the CC model with $O(1)$ RMA complexity. Their algorithms use a `fetch_and_store` object to order passages. When a process $p$ invokes the trying protocol, it performs `fetch_and_store` on a shared variable, Tail, with its ID, $p$. Hence, Tail always contains the ID of the process that was last enqueued. If process $q$ is the predecessor of $p$, then $p$ gets $q$’s ID from Tail when it performs the `fetch_and_store` operation. Then, $p$ waits for a shared variable associated with $q$ to be changed. When $q$ exits the critical section, it updates its associated shared variable to allow $p$ to enter the critical section.

The first local-spin queue-based mutual exclusion algorithm on the DSM model was presented by Mellor-Crummey and Scott [MCS91]. It uses `read`, `write`, `fetch_and_store`, and `compare_and_swap` operations. As in Graunke and Thakkar’s algorithm and Craig’s algorithm, each process performs `fetch_and_store` on Tail with its ID at the beginning of the trying protocol. Instead of waiting for the shared variable associated with its predecessor, a process communicates its ID to its predecessor and then spins on a local variable. An exiting process changes its successor’s spin variable if it gets the ID of its successor in the exit protocol. Otherwise, there are two possible cases: the exiting process does not have a successor until after it finishes the exit protocol, or its successor is already enqueued but has not communicated its ID to the exiting process. To distinguish these cases, an exiting process performs `compare_and_swap` on Tail with its own ID and NIL as an input. If Tail still points to the exiting process, the `compare_and_swap` operation succeeds and resets Tail to NIL. This allows the
next process to enter the critical section without waiting. If Tail does not point to the exiting process, the \texttt{COMPARE\_AND\_SWAP} fails. In this case, the exiting process waits until its successor has communicated its ID. The only problem with this algorithm is that it contains a waiting period in the exit protocol and, hence, does not satisfy the bounded exit property.

Craig \cite{Cra93} tried to eliminate the \texttt{COMPARE\_AND\_SWAP} operation and the waiting period in the exit protocol in Mellor-Crummey and Scott’s algorithm, but his algorithm was incorrect. He uses another \texttt{FETCH\_AND\_STORE} object for hand-shaking, which contains two different fields: one for the state of a predecessor and the other for the ID of a successor. However, his algorithm causes deadlock when a process that returned to the remainder section re-enters the trying protocol and initializes its shared variable, which the successor of its previous passage might access \cite{Lee03}.

In my Master’s thesis \cite{Lee03}, I developed a local-spin mutual exclusion algorithm for the DSM model using only \texttt{FETCH\_AND\_STORE} and registers based on ideas in Craig’s paper. My idea is to assign two different identities for each process, which eliminates the bad situation in Craig’s algorithm. There are three more queue-based local-spin mutual exclusion algorithms using only \texttt{FETCH\_AND\_STORE} and registers in my Master’s thesis. They slightly differ in terms of the number of busy-waiting periods, the size and number of \texttt{FETCH\_AND\_STORE} objects, and total space used. However, all have $O(1)$ RMA complexity and satisfy the FCFS property.

Recently, Golab \cite{Gol10} introduced a new object called \textit{MutexQueue} and showed that most queue-based mutual exclusion algorithms can be viewed as instances of a generic algorithm based on \textit{MutexQueue} and implementations of \textit{MutexQueue}. \textit{MutexQueue} is a queue-like abstract data type that supports \texttt{ENQUEUE}, \texttt{ISHEAD}, and \texttt{DEQUEUE}. The state of \textit{MutexQueue} is an ordered pair $(Q, V)$, where $Q$ is a sequence of distinct process IDs and $V$ is a subset of $Q$. A process in $Q$ becomes a member of $V$ when it writes its ID to a shared variable that its predecessor can access.
When a process performs `enqueue`, its ID is appended to the end of the queue. When a process performs `isHead`, its ID becomes a member of $V$. It returns `true` if and only if the process is at the head of the queue. When a process performs `dequeue`, its ID is removed from the head of the queue. If its ID has a successor, the ID of its successor is returned. A process cannot perform `enqueue` if it is already in the queue, cannot perform `isHead` if it is not in the queue or is already in $V$, and cannot perform `dequeue` unless it is at the head of the queue and is in $V$.

In his mutual exclusion algorithm, each process begins the trying protocol by performing `enqueue` and `isHead`. If its ID is at the head of the queue, it enters the critical section. Otherwise, it waits for its local spin variable to be changed. In the exit protocol, each process performs `dequeue`. If its ID has a successor, it changes the spin variable of its successor to allow the process with that ID to enter the critical section. He also gave two provably correct implementations of MutexQueue, one using only registers and `fetch_and_inc` and the other using only registers and `fetch_and_store`.

In both tree-based and queue-based mutual exclusion, designing a local-spin algorithm for the CC model is much simpler than for the DSM model. This raises the question of whether there exists a way to transform a local-spin algorithm from the CC model to the DSM model. Taubenfeld [Tau04] and Kim and Anderson [KA02] presented local-spin algorithms in the DSM model by modifying local-spin algorithms in the CC model. However, their modifications are not general and may increase the RMA complexity if applied to other algorithms. To achieve a general transformation from the CC model to the DSM model without increasing the RMA complexity, a waiting process $p$ must tell its ID to a process that will change the spin variable on which $p$ is waiting. The main difficulty of this strategy is to avoid a race between these two processes.

I presented two such transformations for mutual exclusion [Lee05]. My first transformation is simpler. It uses a `test_and_set` object for handshaking between two processes. My second transformation uses only registers. It accomplishes handshaking using two
waiting periods and $O(N)$ additional spin variables for each process. I defined a certain class of local-spin mutual exclusion algorithms in the CC model to which my transformations can be applied without increasing the RMA complexity by more than a constant factor.

### 3.2 Lower bounds for mutual exclusion

The difficulty of achieving a local-spin mutual exclusion algorithm using only registers was first addressed by Cypher [Cyp95]. In that paper, Cypher proved that any mutual exclusion algorithm using only \texttt{READ}, \texttt{WRITE}, and comparison-based operations such as \texttt{COMPARE\_AND\_SWAP} and \texttt{TEST\_AND\_SET} requires $\Omega\left(\frac{\log \log N}{\log \log \log N}\right)$ RMAs per passage under the CC or DSM model. This lower bound was improved by Anderson and Kim [AK01] to $\Omega\left(\frac{\log N}{\log \log N}\right)$. Later, Fan and Lynch [FL06] showed an $\Omega(\log N)$ lower bound for mutual exclusion in the state change cost model, in which processes can perform \texttt{READ}, \texttt{WRITE}, and comparison-based operations.

Golab, et al. [GHHW07] showed how to simulate comparison-based operations and \texttt{LL/SC} from \texttt{READ} and \texttt{WRITE} increasing the RMA complexity by only a constant factor. This result implies that, for any mutual exclusion algorithm using comparison-based operations, \texttt{LL/SC}, \texttt{READ} and \texttt{WRITE}, there exists a mutual exclusion algorithm using only \texttt{READ} and \texttt{WRITE} that has the same RMA complexity. Note that the same reduction does not apply to abortable mutual exclusion, since their simulation introduces a waiting period that does not support aborts. Attiya, et al. [AHW08] presented an $\Omega(\log N)$ lower bound proof for mutual exclusion in the DSM or CC model using \texttt{READ} and \texttt{WRITE}. This result together with [GHHW07] shows that, in order to achieve $O(1)$ RMA complexity in the CC or DSM model, a mutual exclusion algorithm must use other operations in addition to comparison-based operations, \texttt{LL/SC}, \texttt{READ} and \texttt{WRITE}. 
3.3 Local-spin abortable mutual exclusion algorithms

Lamport’s bakery algorithm can easily be converted to an abortable mutual exclusion algorithm. When a process wants to abort, it just performs the exit protocol and returns to the remainder section. Although this algorithm is local-spin in the CC model, the abortable version of the algorithm is not. The reason is that, while a process \( p \) is spinning on \( Doorway[i] \) or \( Ticket[i] \), a process \( i \) may abort and re-invoke the trying protocol an unbounded number of times, which updates \( Doorway[i] \) and \( Ticket[i] \) an unbounded number of times. This may cause an unbounded number of cache misses during \( p \)'s waiting period, so the RMA complexity becomes unbounded.

The DSM version of the bakery algorithm in Figure 2 is still local-spin and has the same RMA complexity, even though processes are allowed to abort. However, Taubenfeld’s method to bound the values of the tickets in the bakery algorithm cannot be used when processes are allowed to abort. A process that aborts must not change the color of tickets that are dispensed, because the FCFS property could be violated. But a process may abort and re-invoke the trying protocol an unbounded number of times between two times the critical section is finished, so the value of its ticket can grow without bound.

Scott and Scherer [SI01] first proposed two local-spin mutual exclusion algorithms that allow waiting processes to abort. Their first algorithm, which is local-spin in the CC model, is based on Craig’s mutual exclusion algorithm [Cra93]. Their second algorithm, which is based on Mellor-Crummey and Scott’s algorithm [MCS91], is local-spin on the DSM model. They use registers, \texttt{COMPARE\_AND\_SWAP}, and \texttt{FETCH\_AND\_STORE} objects. The RMA complexity of their algorithms is \( O(N) \) and, when no processes abort, each process performs \( O(1) \) RMAs. However, their abort protocol contains a waiting period in which an aborting process performs handshakes with its predecessor and successor in the queue. Hence, these algorithms do not satisfy the bounded abort property.

Later, Scott [Sco02] eliminated this waiting period in the abort protocol: He presented local-spin abortable mutual exclusion algorithms in which a process aborts within a
bounded number of its own steps. Specifically, when a process aborts, it marks its node in the queue as aborted and, eventually, its successor removes the node in its trying protocol. When no processes abort, each passage performs only a constant number of RMAs in the trying protocol. However, when two or more processes repeatedly abort and re-enter the trying protocol without removing their nodes from the queue, the length of the queue may become unbounded. Hence, these algorithms use unbounded space. The number of RMAs a process performs in the trying protocol can be as large as the number of consecutive times processes began the trying protocol immediately beforehand and subsequently aborted [Sco02, Jay03]. This can be arbitrarily large, since a process can repeatedly enter the trying protocol and abort. Note that, in such a case, the number of RMAs performed by a process is unbounded even though it is local-spinning in its busy-waiting periods. However, the bad situation is only achieved when each passage that aborts decides to do so before its predecessor begins the abort protocol.

Jayanti [Jay03] presented the first local-spin abortable mutual exclusion algorithm with bounded space and bounded RMA complexity. His algorithm is based on the \textit{process-priority-queue}, an abstract data type he defined in a previous paper [Jay02]. A process priority queue allows a process to append or delete a node. It also supports an operation \textit{findmin} that returns the minimum node among the nodes in the queue. Only the owner of a node can append or delete the node, but any process can perform \textit{findmin}. Jayanti presented an implementation of a process-priority queue using LL/SC and registers. His implementation is based on a tree, and its worst case RMA complexity is $\Theta(\log N)$.

To solve abortable mutual exclusion, Jayanti uses a process-priority-queue, LL/SC objects, and registers. In his algorithm, each process gets a distinct time stamp at the beginning of the trying protocol, and inserts a node with its time stamp as its priority into the process-priority-queue. The process with the highest priority (earliest timestamp) enters the critical section first. During the trying, exit and abort protocols, each process
performs a \textit{findmin} operation to find the process with the highest priority in the process-priority-queue, if the critical section is available. Then, it enables the process it found to enter the critical section by changing the spin variable of that process. Although multiple processes might find the same process in the process-priority-queue, only one process should change the spin variable of the found process. Thus, the change of that variable is performed using \texttt{LL/sc} operations. The resulting algorithm has $\Theta(\log N)$ RMA complexity and uses $\Theta(N)$ shared variables. Since time stamps grow without bound, each node has unbounded size.

Since any mutual exclusion algorithm using only registers and \texttt{LL/sc} requires $\Omega(\log N)$ RMAs for each entry to the critical section [AHW08], and since mutual exclusion is a special case of abortable mutual exclusion, Jayanti’s algorithm is optimal in terms of the RMA complexity. Compared with Scott’s algorithm, the worst case RMA complexity of Jayanti’s algorithm is better. However, if the number of consecutive aborts is $o(\log N)$, then each process performs fewer RMAs in Scott’s algorithm than in Jayanti’s algorithm.

### 3.4 \textit{k}-exclusion and group mutual exclusion

The \textit{k}-exclusion problem was first studied by Fischer et al. [FLBB79]. Their algorithm tolerates up to $k - 1$ process crashes but is not local-spin. Anderson and Moir [AM97] developed a local-spin \textit{k}-exclusion algorithm using \texttt{FETCH\_AND\_INC} and \texttt{FETCH\_AND\_DEC}, which has $\Theta(N - k)$ RMA complexity. They showed how to convert an $(\ell - 1)$ process \textit{k}-exclusion algorithm to an $\ell$ process \textit{k}-exclusion algorithm. Then, they applied this conversion recursively to achieve a $N$ process \textit{k}-exclusion algorithm. Note that a \textit{k} process \textit{k}-exclusion algorithm is trivial since all processes can enter the critical section without waiting. The resulting algorithm tolerates up to $k - 1$ process crashes. In [Dan10], Danek presented a local-spin \textit{k}-exclusion algorithm using only registers that tolerates up to $k - 1$ process crashes.
Danek and Hadzilacos [DH04] presented a group mutual exclusion algorithm with concurrent entering by giving a transformation from an FCFS abortable mutual exclusion algorithm to a group mutual exclusion algorithm. Combined with Jayanti’s abortable mutual exclusion algorithm, their transformation gives the first local-spin group mutual exclusion. Their algorithm takes $\Theta(N)$ RMAs per passage in the CC or DSM model.

3.5 Memory reclamation methods

In real-life systems, only a finite amount of memory is available. However, an algorithm may use more space than is available on a given system. To enable the algorithm to work in this system, it is necessary to recycle memory that is no longer being used. This procedure is called memory reclamation.

In Chapter 7, we present a new algorithm using unbounded space and, in Chapter 8, we use a new memory reclamation method to bound the space used by our algorithm. In this section, we survey the main issues and existing methods for memory reclamation, and explain why none of them are suitable for use in our algorithm.

3.5.1 Overview and definitions

First we describe how memory is allocated to objects. A system maintains a pool of the addresses of unused memory locations. When a process creates an object, the system allocates a set of contiguous memory locations for the object. This is called a node. To allocate a node, the system finds memory locations suitable for the object, removes the addresses of these memory locations from the pool, and gives the address of the first memory location to the process. The process stores this address as a pointer. This procedure is called memory allocation.

A node is garbage when it is no longer being used by any process and cannot be used by any process before it is reclaimed. Since the size of memory is limited, it is important
to reuse the memory in garbage nodes. This is called memory reclamation.

It is useful to classify the nodes used in an algorithm. A static node is a node that is never reclaimed during any execution of the algorithm, for example, a variable containing a pointer to the head of a linked list. A dynamic node is a node that can be reclaimed while the algorithm is running, for example, an element of a linked list. Thus, only dynamic nodes can be garbage.

A process can access a dynamic node only if it has a pointer to the node. A local pointer is the address of a node stored locally at a process, which has exclusive right to access the pointer. A shared pointer is the address of a node that is stored in a node. A shared pointer is also called a reference. If the node that stores a shared pointer is static, a process can access the shared pointer without having a pointer to the node. If the node that stores a shared pointer is dynamic, a process can access the shared pointer only if it has a local pointer to the node.

If there is no shared or local pointer to a dynamic node, no process can access the node, so it becomes garbage. More precisely, all nodes are categorized as either reachable or unreachable. A node is reachable if it is a static node or some process has a local pointer to the node. Moreover, if a reachable node has a pointer to a dynamic node, then the latter node is also considered reachable. Then, we define a node to be garbage if and only if it is unreachable.

There are two methods for memory reclamation: memory deallocation and maintaining free-lists. When a node is deallocated, the addresses of its memory locations are inserted into the pool. Afterwards, they can be reallocated, either together or separately, for another purpose. Deallocated memory locations can even be used by another algorithm. Memory deallocation enables the memory space used by an algorithm to shrink.

Alternatively, an algorithm can maintain one or more free-lists of garbage. When a node becomes garbage, it is not deallocated, but, instead, is inserted into a free-list. When a process wants to create a new node, it may use a node in a free-list, instead
of performing memory allocation. The nodes in a free-list cannot be reused by another algorithm, which may result in less efficient use of memory than deallocating nodes.

*Premature reclamation* occurs when a node that is not garbage is reclaimed. This may cause problems. For example, suppose that, in an algorithm using two queues, the last node of the first queue is removed, the node is reclaimed, and then the memory used by the node is reused as the last node of the second queue. If some process wants to append a new node to the first queue, but does not know that the last node of the first queue has been reclaimed and reused, then it may mistakenly append a new node to the second queue.

Another problem is that a node which represented a pointer may now represent an integer. If some process mistakenly still treats the node as a pointer and accesses the memory location pointed to by the node, then a system error may occur if it is not permitted to access that location. This problem can be avoided by using separate free-lists for different object types. Premature reclamation may cause more serious problems since a memory location that was reclaimed prematurely can contain any value. Consequently, an algorithm may behave in an unexpected manner or an undetected error might occur.

It is difficult for a process to determine whether a node is garbage since it cannot access the local pointers of other processes. Three methods have been proposed to correctly address this problem: garbage collection, hazard pointers, and reference counting. These are discussed in the remainder of the chapter. At the beginning of Chapter 8, we will explain why none of these methods is suitable for our application.

### 3.5.2 Garbage collection

In garbage collection, garbage is collected by one or more special processes called *garbage collectors*. These processes can access all shared and local pointers.

The simplest example of garbage collection is the *mark-and-sweep* algorithm: Each node has a Boolean flag that indicates if the node is reachable. A garbage collector
first sets the flag of all reachable nodes using a graph traversal algorithm such as depth
first search, starting from each static node and each node to which some process has
a local pointer. Then, all nodes are scanned, during which all unreachable nodes are
reclaimed and the flags of all reachable nodes are reset. This method has several major
disadvantages. First, other processes must stop while garbage collection is going on and
no garbage is reclaimed until garbage collection is performed. Also, the garbage collectors
must access the entire memory, which takes a lot of time.

There are many other garbage collection algorithms that try to overcome some of these
disadvantages [PS95, AR98]. For example, in the tri-color garbage collector algorithm,
proposed by Dijkstra et al. [DLM+78], each node has one of three colors (white, gray
and black), instead of one Boolean flag as in the mark-and-sweep algorithm. Initially,
all static nodes and nodes that are pointed to by local pointers are gray and all other
nodes are white. When a garbage collector finds a gray node that points to a white
node, it changes the white node to gray. If a gray node does not point to any white
node, it becomes black. When there are no gray nodes remaining, all reachable nodes
are black and all unreachable nodes are white. Then all white nodes are reclaimed. In
this algorithm, garbage collection may be performed concurrently with other operations.

3.5.3 Hazard pointers

The hazard pointer algorithm, proposed by Michael [Mic04], does not have special pro-
cesses dedicated to memory reclamation. Instead, any process can reclaim garbage when
it detects it. In this method, if a process has a local pointer to a node and will use
the pointer later, then it stores the address of the node in a shared object as a hazard
pointer. Each process maintains a list of its hazard pointers. This list is updated only by
the process, but can be read by other processes. When a process no longer uses a node, it
removes its pointer to the node from its list of hazard pointers. Then it scans the hazard
pointers of all other processes. If there is no remaining hazard pointer to the node, the
node can be safely reclaimed. The author claims that, in most existing lock-free data structures, each process requires only a small number of hazard pointers. However, even if each process has $O(1)$ hazard pointers, it takes $\Omega(N)$ time to determine if some node is garbage.

Suppose that, at every point in every execution, each process uses at most $h$ hazard pointers. If each process has a pool of $2hN$ nodes, then garbage nodes can be reclaimed in $O(1)$ amortized time: When its pool is empty, a process scans the hazard pointers of every process, which takes time $O(hN)$. At least $hN$ of its nodes have no hazard pointers pointing to them and the process can put these nodes back into its pool.

Herlihy et al. [HLM02] also proposed a similar method. They defined the repeat offender problem, which captures the memory reclamation problem in lock-free data structures, and then presented a solution for the problem. Their solution is similar to hazard pointers.

### 3.5.4 Reference counting

In reference counting methods, each node stores an upper bound on the number of references (i.e., shared pointers) to it. This is called a reference count. When the reference count of a node becomes zero, the node can be reclaimed. Reference counting allows any process to reclaim garbage without scanning many nodes.

Reference counting was first proposed by Collins [Col60] for an implementation of the Lisp programming language in a system with a single process. The most notable disadvantage is that, when there is a cycle of shared pointers among unreachable nodes, none of them can be reclaimed. Also, each node stores a count, which increases the space used by the node.

A naive algorithm for reference counting is as follows: The reference count of a node is incremented before a new pointer to the node is created, and the reference count of a node is decremented after an existing pointer to the node is deleted. Changing a shared
pointer that points to a node $A$ to point to a node $B$ is equivalent to creating a shared
pointer to $B$ and then deleting a shared pointer to $A$. This method preserves the reference
count invariant: in every configuration, the reference count of a node is an upper bound
on the number of references to the node.

The naive algorithm works in single process environments, but premature reclamation
can occur in multi-process environments. For example, consider a pointer $x$ to a node $A$
whose reference count is 1. Suppose that some process $P$ reads $x$, creates a local pointer
to $A$, and wants to store this in another shared pointer $y$. Then, process $Q$ changes the
value of $x$, decrements $A$’s reference count to 0, and reclaims $A$. If $P$ tries to increment
$A$’s reference count, premature reclamation occurs.

To solve this problem, a number of different methods have been proposed. A trivial
solution is to read a shared pointer $x$ and increment the reference count of the node to
which $x$ points in one atomic step. This solution requires a powerful atomic operation
that accesses two different nodes at once. However, this operation is not supported
in real-life systems. In the rest of this chapter, we survey existing reference counting
methods that use more common operations.

### 3.5.5 Lock-free reference counting using DCAS

Detlefs et al. [DMMJ01] proposed a lock-free reference counting method, that uses
double compare and swap (or DCAS). This operation, which is supported in some
systems such as the Motorola 68040 [DFG+00], allows a process to access two nodes at
the same time. A DCAS operation atomically updates the values of two objects only when
each of the two objects has the same value as each of two expected values. A formal
definition of DCAS is as follows:

\[
\text{DOUBLE\_COMPARE\_AND\_SWAP} \ (V, W, x, y, a, b) \ \{
\text{if } V = x \text{ and } W = y \text{ then } V := a \\
\quad W := b
\}
\]
Their algorithm creates a shared pointer to a new node and deletes a shared pointer the same way as the naive algorithm does. However, when a process \( P \) reads a shared pointer \( x \) that points to \( A \) and creates a new shared pointer to \( A \) (i.e., copying a shared pointer \( x \)), it uses \texttt{dcas}. More precisely, \( P \) reads \( x \), reads \( A \)'s reference count, and then performs a \texttt{dcas} operation on \( x \) and \( A \)'s reference count. If \( x \) or \( A \)'s reference count is different than what \( P \) read, the \texttt{dcas} fails and \( P \) can try again. If both \( x \) and \( A \)'s reference count are the same as what \( P \) just read, the \texttt{dcas} succeeds, \( x \)'s value is unchanged and \( A \)'s reference count is incremented. Then, \( P \) creates a new shared pointer to \( A \).

In this algorithm, a process can access all fields of a node, other than its reference count, only after it increments the reference count of the node and before it decrements the reference count of the node. Hence, it is guaranteed that the node containing a shared pointer \( x \) is not reclaimed while \( x \) is being copied. To see why, suppose that the shared pointer \( x \) is stored in a node \( y \). If \( y \) is a static node, then \( y \) is not reclaimed until the algorithm ends. If \( y \) is a dynamic node, then between when \( P \) last read \( x \) and when \( P \) performs the \texttt{dcas}, \( y \)'s reference count is greater than 0. This is because \( P \) incremented \( y \)'s reference count before reading \( x \) and does not decrement \( y \)'s reference count during this period. Thus \( y \) is not reclaimed during this period.

The correctness of the algorithm follows from this observation and the semantics of the \texttt{dcas} operation. Creating a pointer to a new node and deleting a shared pointer use the naive algorithm, which preserves the reference count invariant. When a process copies a shared pointer \( x \) that points to \( A \), there are three cases. If the \texttt{dcas} fails, then this does not change \( A \)'s reference count or create a shared pointer to \( A \), so the reference count invariant still holds. If \( x \) and \( A \) are not changed between when \( P \) last read \( x \) and when \( P \) performs a \texttt{dcas}, the \texttt{dcas} succeeds, so \( A \)'s reference count is incremented. Then, a pointer to \( A \) is created. Hence, the reference count invariant holds after copying
the pointer. Finally, consider the case that, between when \( P \) last read \( x \) and when \( P \) performs a DCAS, either \( x \) or \( A \) was changed, but was then changed back to the value that \( P \) last read. By the observation above, the node from which \( P \) read \( x \) is not reclaimed during the copying of \( x \). Since \( x \) points to \( A \) when \( P \) performs the DCAS, \( A \) is being used by the same algorithm and hence the reference count invariant holds prior to the DCAS, even though \( A \) may have been reclaimed and reused during this period of time. Since \( A \)'s reference count is incremented by the DCAS and then a new pointer to \( A \) is created, the reference count invariant holds after copying the pointer.

The main weakness of this algorithm is that it is not wait-free. Also, few systems support DCAS. Another issue is that the algorithm allows premature reclamation. However, if premature reclamation occurs, then any access to a reclaimed node is done by an unsuccessful DCAS, which does not change its value. More specifically, when \( P \) copies a pointer \( x \) that points to \( A \), the DCAS by \( P \) can access the memory location allocated for \( A \) after \( A \) has been reclaimed. Since the memory for \( x \) has not been reclaimed during the copying of \( x \), \( x \) no longer points to \( A \), so this access does not change the contents of \( A \). Some operating systems may prohibit a process from accessing memory locations that have been deallocated until they are re-allocated. In such systems, the algorithm must use a free-list for reclaimed nodes.

### 3.5.6 Lock-free reference counting using a free-list

Another reference counting method, proposed by Valois [Val95], does not use an operation that accesses two nodes at a time. Instead, this algorithm always maintains a free-list for reclaimed nodes. Creating a shared pointer to a new node or deleting a shared pointer is the same as in the naive algorithm. When a process \( P \) wants to copy a shared pointer \( x \) that points to \( A \), \( P \) increments the reference count of \( A \) and then checks \( x \) again. If \( x \) is changed, then \( P \) decrements \( A \)'s reference count and tries again. If \( x \) is not changed, then \( P \) creates a new shared pointer to \( A \).
To indicate whether a node is in the free-list, each node has a `test_and_set` object, `flag`. The flag of a node \( A \) is set to one before \( A \) is inserted in the free-list, and set to zero after \( A \) is removed from the free-list. When the reference count of \( A \) becomes zero, a process performs a `test_and_set` on \( A \)'s flag and, if the `test_and_set` succeeds, it moves \( A \) to the free-list.

In this algorithm, \( A \)'s reference count can be changed even after it has been reclaimed. Thus, the memory locations for \( A \) cannot be deallocated, so the algorithm uses a free-list. If a process fails immediately after successfully performing the `test_and_set`, the node will never be moved to the free list.

A more serious problem, observed by Michael and Scott [MS95], is that a race condition can generate an incorrect execution of this algorithm: Suppose that, while \( A \) is in the free-list, \( P \) increments \( A \)'s reference count and then decrements \( A \)'s reference count back to 0. \( P \)'s next step is performing a `test_and_set` to \( A \)'s flag to see if \( A \) is already in the free-list. However, suppose that this step is delayed. Since \( A \) is in the free-list and its reference count is zero, \( A \) can be re-used, say, by another process \( Q \), which moves \( A \) out of the free-list and resets \( A \)'s flag. Now \( P \) performs its `test_and_set` operation, which succeeds. Thus \( P \) will move \( A \) to a free-list. However, this causes an error, since \( Q \) is currently using \( A \). Michael and Scott corrected this problem using a non-standard atomic operation (applied to one node): \( P \) atomically decrements \( A \)'s reference count and sets \( A \)'s flag. They also gave a lock-free implementation of this operation using `cas`.

### 3.5.7 Reference counting methods using proxy incrementing

There are several reference counting methods in which the reference count of a node is not incremented by a process that creates a new pointer to the node. Instead, incrementing the reference count of a node is always done by a process that has been given the privilege to do so. We call this `proxy incrementing`. In general, when a process creates a pointer to a node \( A \), it only increments \( A \)'s reference count if it has the privilege to do so. Otherwise,
it asks another process which has this privilege to increment \( A \)'s reference count on its behalf.

Lermen and Maurer [LM86]'s \textit{distributed reference counting} is such a method for a message passing model with FIFO links. In this algorithm, each node has a unique owner. Only the owner has the privilege of incrementing and decrementing its reference count. When a process \( P \) wants to create a shared pointer to a node \( A \) owned by process \( R \), \( P \) first sends \( inc(P, A) \) to \( R \). After receiving \( ack \) from \( R \), \( P \) creates a shared pointer to \( A \). When \( P \) wants to delete a shared pointer to a node \( A \) owned by process \( R \), \( P \) sends \( dec(A) \) to \( R \) and then deletes the shared pointer without needing to receive an \( ack \) from \( R \).

When \( P \) wants to copy a shared pointer \( x \) that points to node \( A \), \( P \) first sends a message, \( copy(x) \), to the owner \( Q \) of \( x \). Then, \( P \) waits until it receives the address of \( A \) from \( Q \) and an \( ack \) message from the owner \( R \) of \( A \). If \( Q \) receives a \( copy(x) \) message from \( P \) and \( Q \) has not deleted \( x \), it sends the value of \( x \) (i.e., the address of \( A \)) to \( P \), and then immediately sends an \( inc(P, A) \) message to the owner \( R \) of \( A \). If \( Q \) has already deleted \( x \), then \( Q \) sends a \( fail \) message to \( P \). When \( R \) receives an \( inc(P, A) \) message, it increments \( A \)'s reference count and sends an \( ack \) message to \( P \). \( P \) creates a shared pointer to \( A \) only after it receives an \( ack \) message from \( R \). Hence, \( A \)'s reference count is incremented before a shared pointer to \( A \) is created. This preserves the reference count invariant. Note that \( Q \) does not send a \( dec(A) \) message to the owner of \( A \) between sending the address of \( A \) to \( P \) and sending an \( inc(P, A) \) message to the owner of \( A \). Hence if \( P \) copies \( x \) and then \( Q \) deletes \( x \), \( A \)'s reference count is incremented before it is decremented, given that all links are FIFO.

This algorithm has several disadvantages. It forces processes to wait for messages, so it is not wait-free. Moreover, the owner of a node has to perform a lot of work if many shared pointers to the node are created, i.e., the owner becomes a bottleneck. Finally, if used in a shared memory model, each owner would have to continually check to see if it
has been sent any messages.

Bevan [Bev87]'s weighted reference counting is another example of proxy incrementing. In this algorithm, each pointer has an associated positive integer, called a *weight*. Each node has a count, which is set to a large value by its creator and is only decremented. Thus, it can be said that only the creator has the privilege to increase the count.

When a process creates the first shared pointer to a new node $A$, both $A$’s count and the weight of this pointer are set to some large value, usually $2^k$ for some integer $k$. When another process reads a shared pointer $x$ to $A$ and creates a new pointer $y$ to $A$, half of $x$’s weight is given to $y$. After a process deletes a shared pointer $x$ to $A$, it subtracts $x$’s weight from $A$’s count. Hence, $A$’s count is always bounded below by the sum of the weights of the shared pointers to $A$.

If the weight of a shared pointer becomes one, it cannot be copied further. Hence, if the weight of a pointer is at most $2^k$, it can be copied at most $k$ times. Bevan solved this problem by creating a new pointer between the existing shared pointer and the node. When the weight of a shared pointer $x$ that points to $A$ is one and some process wants to copy it, the process creates a new node $B$ holding a new pointer $y$ that points to $A$. The process then sets $y$’s weight to one, and makes $x$ point to $B$ instead of $A$. Both $B$’s count and $x$’s weight are set to $2^k$ so that $x$ can be copied further. This method generates extra nodes and pointers. Also, when a process wants to access node $A$, it may have to follow a chain of pointers, which increases the number of RMAs it performs.

Goldberg [Gol89] presented generational reference counting, which also uses proxy incrementing. In this algorithm, each pointer has a generation number in $\{0, \ldots, k - 1\}$ and a count in $\mathbb{N}$, in addition to an address, and each node $A$ has an array of reference counts, $R_A[0 \ldots k - 1]$, where $k$ is a predetermined number. The first pointer to a node has generation number zero. When a new pointer is created as a result of copying a pointer with generation number $i$, the generation number of the new pointer is $i + 1$. The generation number of a pointer is not changed until the pointer is deleted.
This algorithm maintains the following invariant: for all $0 \leq i \leq k - 1$, $R_A[i]$ plus the sum of the counts in all pointers to $A$ with generation $i - 1$ is always greater than or equal to the number of pointers to $A$ with generation $i$. Note that there are no pointers to $A$ with generation $-1$, so $R_A[0]$ is at least the number of pointers to $A$ with generation one. This implies that, if $R_A[0] = \ldots = R_A[i] = 0$, then there are no pointers to $A$ with generation at most $i$. Hence, when all elements are zero, there are no pointers to $A$, so $A$ can be reclaimed.

When a process creates the first pointer $x$ to $A$, it sets $R_A[0]$ to one, $x$’s generation number to zero and $x$’s count to zero. A process can only copy a shared pointer with generation number less than $k - 1$. When a process reads a shared pointer $x$ to $A$ with generation number $i < k - 1$ and creates another shared pointer $y$ to $A$, it reads all the fields of $x$ and atomically increments $x$’s count. Then it initializes $y$’s generation number to $i + 1$, sets $y$’s count to zero, and stores the address of $A$ in $y$. When a process deletes a shared pointer $x$ to $A$, where $x$’s generation number is $i < k - 1$, the process atomically decrements $R_A[i]$ and adds $x$’s count to $R_A[i + 1]$. When a process deletes a shared pointer $x$ to $A$, where $x$’s generation number is $k - 1$, the process simply decrements $R_A[k - 1]$.

A weakness of this algorithm is that each node $A$ has a big array $R_A$ for its reference counts all stored in one word. Alternatively, $R_A$ can be represented using $k$ words, provided $R_A[i + 1]$ is incremented before $R_A[i]$ is decremented. Furthermore, the space used by each node depends on how large the value $k$ is set. The algorithm does not allow a pointer with generation $k - 1$ to be copied further. Hence, if $k$ is smaller, the algorithm has more restricted application. The author proposed a method to increase the size of $R_A$ when a pointer with generation $k - 1$ is copied. However, this method requires additional indirection which, like Bevan’s weighted reference counting, is inefficient. Moreover, if $k$ is large, but there are only a small number of pointers to $A$ at any one time, the algorithm uses more space than necessary.
3.5.8 Specialized methods

In this section, we describe three algorithms to bound space for specific dynamic data structures: two universal constructions and one algorithm for the collect problem.

Herlihy’s universal construction [Her91] uses a doubly-linked list of nodes. Each node has a sequence number and a count. When a node is successfully appended, its sequence number becomes one greater than its predecessor in the list. There are two shared arrays, each with size $N$: $Announce$ is an array of pointers to the node each process is appending or last appended, and $Head$ is an array of pointers to the node with highest sequence number that each process has accessed.

When a process $P$ appends a new node to the linked list, $P$ first creates a new node with sequence number 0, and announces it by updating $Announce[P]$. Then, $P$ finds the maximum of the sequence numbers of the nodes pointed to by the $Head$ array. $P$ will try to append a new node following the node with this sequence number. Note that this node was the last node in the linked list at some time after $P$ started its operation. To achieve wait-freedom, $P$ helps other processes append their nodes. If $s$ is the sequence number of the last node in the list, then $P$ tries to append the new node announced by the process with ID $s \mod N$, if such a node exists. Otherwise, $P$ tries to append its own new node. $P$ does this until its new node is appended.

To see how long this takes, consider the sequence number $i$ of the node being appended when $P$ announces its new node. Among $\{i + 1, \ldots, i + N\}$, there exists $j$ such that $j \mod N = P$. If $P$’s node is not given a sequence number in $\{i, \ldots, j - 1\}$, then all processes that try to append the node with sequence number $j$ will try to append $P$’s node. Hence, $P$ tries to append at most $N + 1$ nodes. After $P$’s new node is appended, it increments the counts of the $N + 1$ consecutive nodes preceding its new node. When the count of a node becomes $N + 1$, the node can be reclaimed.

In this algorithm, when each process reads the sequence numbers of the nodes pointed to by the $Head$ array, some of these nodes may have already been reclaimed. This might
be a problem if reclaimed nodes are deallocated. If so, reclaimed nodes should be stored in a free-list.

Herlihy presented another universal construction [Her90, Her93], which also uses a counting method for memory reclamation. The algorithm in [Her93] uses $N + 1$ nodes. Each process has a pointer to a different node and there is a global pointer, $root$, that points to the remaining node. To update a shared object, a process $P$ first performs $\text{LL}$ on $root$ and reads the value of the node pointed to by $root$. Then $P$ writes the updated value to the node pointed to by $P$’s pointer. Finally it performs $\text{SC}$ to attempt to set $root$ to point to this node. If the $\text{SC}$ operation succeeds, $P$ changes its pointer to the node that $root$ used to point to.

This method allows a process to read a reclaimed node. Suppose process $Q$ performs $\text{LL}$ on $root$, which points to node $x$. Next, suppose process $P$ performs $\text{LL}$ on $root$, creates a new node $y$, successfully performs $\text{SC}$ on $root$, and reclaims $x$. Then, $Q$ reads $x$ after $x$ has been reclaimed by $P$. If $Q$ reads the value of $x$ while $P$ is updating $x$, the value of $x$ may not be consistent because $P$ may need to perform a number of writes to update (a large) node $x$. Although $Q$’s next $\text{SC}$ operation will fail, problems may occur when $Q$ is computing an updated value.

Herlihy suggested two solutions. In both solutions, before $Q$ performs this computation, $Q$ checks if $root$ has been updated by another process since $Q$’s last $\text{LL}$. This can be done using a validate primitive, which returns $\text{TRUE}$ if and only if no process has successfully performed an $\text{SC}$ since $Q$’s last $\text{LL}$. If validate is not available, a counting method is used to check the validity of the node pointed to by $root$. Each node has two counters, $C[0]$ and $C[1]$. When a process reads a node, it reads $C[1]$ before reading the node and reads $C[0]$ after reading the node. When a process writes to the node, it increments $C[0]$ before the write and increments $C[1]$ after the write. Hence, a reading process sees $C[0] = C[1]$ only if no other process had modified the node while the process was reading the node.
The algorithm in [Her90] uses cas instead of LL/sc. When a process $p$ wants to update $root$ to point to an updated node, it reads $root$ and performs cas on $root$ with the value it last read and the address of the updated node. Note that, if $root$ has been updated and then changed back to the value $p$ last read between $p$'s last read and $p$'s cas operation, the cas still succeeds.

To avoid this situation, each node has a counter, and a node can be reclaimed only when its counter is zero. To update a shared object, process $p$ reads $root$, increments the counter of a node $A$ pointed to by $root$, and reads $root$ again. If $root$ has been changed between these two reads, $p$ aborts the update and decrements $A$’s counter. Otherwise, $p$ creates an updated node $B$ using one of reclaimed nodes, and performs cas on $root$ with the address of $A$ and the address of $B$. Note that, after $p$ increments $A$’s counter, $A$ cannot be reclaimed and reused until after $p$ decrements $A$’s counter. Hence, it is impossible that $root$ has been updated and changed back to the address of $A$ after $p$ last read $root$. Then, finally, $p$ decrements $A$’s counter. If the cas succeeds and $A$’s counter becomes zero, $p$ reclaims $A$.

In this algorithm, each process maintains a free-list of size $N$. Since each process increments the counter of at most one node at a time, there is at least one node in each free-list. Therefore, this algorithm uses $N^2 + 1$ nodes in total.

Herlihy et al. [HLM03] used a reference counting method in their lock-free algorithm for the collect problem. This algorithm uses a doubly-linked list that supports three operations, acquire, delete, and scan. Each node has a flag indicating if the node is owned or deleted. There is a global pointer, $head$, that points to the first node in the list. Each forward pointer has a count.

When a process wants to acquire a new node in the list, it follows the linked list from $head$, incrementing the count of each pointer it reads. If it finds a deleted node, then it acquires this node by changing its flag from deleted to owned using cas. Otherwise, it append a new node to the end of the list using cas. A process can only delete a node it
owns. To do so, it changes the *flag* of the node from owned to deleted and then follows
the linked list back to the beginning, decrementing the counts of all forward pointers it
read. When a process performs scan, it follows the list from beginning to end and reads
the values of all owned nodes. Each time it reaches a forward pointer, it increments the
count of the pointer. After reaching the end of the list, the process follows the linked list
back to beginning of the list, decrementing the counts of all forward pointers it read.

In this algorithm, the count of a forward pointer indicates the number of owned nodes
after the pointer plus the number of processes performing scan that are accessing a node
after the pointer. Hence, if the count of a pointer is zero, then all nodes after the pointer
have been deleted and no process is accessing any of these nodes. Thus, the pointer can
be removed and all nodes after the pointer can be reclaimed. Note that, if each process
owns at most one node, then the length of the list is at most $N$. 
Chapter 4

An Algorithm Using Only Registers

In this chapter, we present the first local-spin abortable mutual exclusion algorithm using only registers. This algorithm also solves other related problems.

4.1 Algorithm

Our algorithm is a surprisingly simple modification of Yang and Anderson’s mutual exclusion algorithm [YA95], which is described in Section 3.1.1. To make this algorithm abortable, we make each two-process mutual exclusion algorithm in the tree abortable. With this change, a process can abort its attempt to enter the critical section any time it is busy-waiting. When a process aborts, it executes the abort protocol of the current node, and then traces the path from that node to its leaf, executing each node’s abort protocol along the way.

Figure 5 shows an instance of the two-process abortable mutual exclusion algorithm at any internal node. This algorithm is the same as Yang and Anderson’s two-process mutual exclusion algorithm, except that a process can abort by executing the exit protocol. Like Yang and Anderson’s algorithm, it has two busy-waiting periods (lines 9 and 12) and a spin variable $Spin[p]$, which has three states: RED, YELLOW, and GREEN. RED indicates process $p$ cannot terminate either busy-waiting period; YELLOW indicates $p$
Figure 5 An Algorithm Using Only Registers (two process version) (Code for process $p$)

shared variables: ($L$ is the set of process IDs that can participate in this algorithm.)

Want: array[0,1] of $L \cup \{NIL\}$, initially NIL
LastIn $\in L$
Spin: array of \{RED, YELLOW, GREEN\} indexed by $L$, initially RED
($Spin[p]$ is located in $p$’s local memory.)

private constant:
$side = 0$ if $p$ is in the left subtree of the node; otherwise, $side = 1$

private variable:
rival $\in L \cup \{NIL\}$

TryingProtocol()
1: $Want[side] := p$;
2: $LastIn := p$;
% Doorway ends
3: $Spin[p] := RED$;
4: rival := $Want[1 – side]$;
5: if rival $\neq NIL$ then
6: if $LastIn = p$ then
7: if $Spin[rival] = RED$ then
8: $Spin[rival] := YELLOW$;
end if
9: while $Spin[p] = RED$ do
10: // (If $p$ wants to abort, goto line 14.)
end while
11: if $LastIn = p$ then
12: while $Spin[p] \in \{RED, YELLOW\}$ do
13: // (If $p$ wants to abort, goto line 14.)
end while
end if
end if

ExitProtocol() / AbortProtocol()
14: $Want[side] := NIL$;
15: rival := $LastIn$;
16: if rival $\neq p$ then
17: $Spin[rival] := GREEN$;
end if

The correctness proof of our algorithm is more difficult than the correctness proof of Yang and Anderson’s algorithm because the identity of $p$’s private variable rival may change repeatedly during one passage due to aborts by waiting processes. In Yang and
Anderson’s algorithm, if process $p$ finishes the doorway before process $q$ does and $p$ sets $rival$ to $q$ in line 4, then $q$ must wait for $p$ to finish the critical section. After finishing the critical section, $p$ sets $rival$ to either $p$ or $q$ in line 15, depending on whether or not $q$ has finished the doorway. However, in our algorithm, $q$ may abort and another process $r$ may invoke the trying protocol, so $p$ may set $rival$ to $r \neq p, q$ in line 15.

In the next section, we prove that our algorithm is correct. In Section 4.3, we analyze the complexity of our algorithm. Finally, in Section 4.4, we present applications of this algorithm to other variants of mutual exclusion.

### 4.2 Proof of correctness

Since the structure of the algorithm is a tournament tree, we prove the correctness of the algorithm by induction on the height of the tree. We can consider each leaf as a one-process mutual exclusion algorithm. The one-process algorithm has no entry or exit protocol, and it consists of only the critical section. It trivially satisfies mutual exclusion and lockout freedom since only one process is involved in the one-process algorithm.

Consider an arbitrary non-leaf node in the tournament tree. We prove that the two-process mutual exclusion algorithm at this node satisfies mutual exclusion and lockout freedom, given, as an induction hypothesis, that the algorithms at all nodes in its left and right subtrees satisfy mutual exclusion and lockout freedom. Specifically, we prove that at most one process can enter the critical section of the node at the same time and that every busy-waiting process at the node eventually enters the critical section if it does not abort.

Each of the following observations and invariants has the form of an implication, in which the antecedent limits both processes to certain parts of the algorithm, and the consequence is a condition on the values of certain variables. We prove that any event that changes the antecedent from false to true makes the consequence true. Furthermore,
the abort event, which changes the next step from line 9 or 12 to line 14, does not change
the truth of the antecedent from false to true. Note that the abort event does not change
any variables. The proofs of these invariants are similar to the proofs of the mutual
exclusion algorithms in [YA95] and [KA06].

When a process $p$’s next event is line $i$, we say $p@i$. If a process $p$ finished line $i - 1,$
but not yet finished or skipped line $j,$ then we say $p@\{i..j\},$ which also means $p@i,$
$p@(i + 1), \ldots, or p@j.$ A process $p$ is in the trying protocol if $p@\{2..12\}$ and is in the
exit protocol or abort protocol if $p@\{15..17\}.$ A process $p$ in the trying protocol has
finished the doorway if $p@\{3..12\}.$ If $p$ is in the critical section, then $p@14.$ Immediately
after $p$ performs line 17 or $p$ performs line 15 with $LastIn = p,$ $p$ is in the remainder
section.

We denote the event in which process $p$ performs line $i$ by $p.i$. We also use $\prec$ to
denote the order of two events in an execution. For example, $p.i \prec q.j$ means that $p$
performs line $i$ before $q$ performs line $j.$ Also, the owner of a private variable is denoted
by a subscript. Note that lines 5, 10, 13 and 16 are not events since they only involve
local variables and, hence, $p@5, p@10 p@13$ and $p@16$ are never true.

Note that, by the induction hypothesis, at most two processes perform the algorithm
in Figure 5 on any specific node at the same time and, when two processes simultaneously
perform the algorithm, one process has 0 for the value of $side$ and the other has 1. The
first two observations follow from these facts. For the rest of the chapter, let $p$ and $q$ be
arbitrary processes, where $side_p = 0$ and $side_q = 1,$ and let $x, y \in \{p, q\}$ be such that
$x \neq y.$

**Observation 4.1.** If $x@\{2..14\},$ then $Want[side_x] = x.$ $Want[side_x] = NIL$ if and only
if there is no process $r$ (including $x$) such that $r@\{2..14\}$ and $side_r = side_x.$

**Observation 4.2.** If both $p$ and $q$ have finished the doorway, but have not subsequently
returned to the remainder section (i.e., $p,q@\{3..17\}$), then $LastIn$ is either $p$ or $q.$
Invariant 4.3. If \( x \in \{6..14\} \) and \( y \in \{3..14\} \), then \( \text{rival}_x = y \) or \( \text{LastIn} = y \).

Proof. While \( x \in \{6..14\} \) and \( y \in \{3..14\} \), the values of \( \text{rival}_x \) and \( \text{LastIn} \) remain unchanged. The antecedent becomes true when \( x \) performs lines 4 and 5 with \( \text{Want}[1 - \text{side}_x] \neq \text{NIL} \) while \( y \in \{3..14\} \) or when \( y \) performs line 2 while \( x \in \{6..14\} \). In the first case, by Observation 4.1, \( \text{Want}[1 - \text{side}_x] = \text{Want}[\text{side}_y] = y \) while \( y \in \{3..14\} \). Hence, \( x \) writes \( y \) to \( \text{rival}_y \) in \( x.4 \), so \( \text{rival}_x = y \) in \( x \in \{6..14\} \). In the second case, \( y \) writes \( y \) to \( \text{LastIn} \) in \( y.2 \), so \( \text{LastIn} = y \) in \( y \in \{3..14\} \).

\( \square \)

Invariant 4.4. If \( x \in \{4..17\} \) and \( y \) has finished the doorway but not subsequently returned to the remainder section (i.e., \( y \in \{3..17\} \)), then \( \text{Spin}[x] \neq \text{GREEN} \) or \( \text{LastIn} = y \).

Proof. While \( x \in \{4..17\} \) and \( y \in \{3..17\} \), the value of \( \text{LastIn} \) remains unchanged and \( \text{Spin}[x] \) cannot be set to \text{GREEN}. If \( x \) performs line 3 while \( y \in \{3..17\} \), then \( \text{Spin}[x] \) is set to \text{RED}, so the consequence holds. If \( y \) performs line 2 while \( x \in \{4..17\} \), then \( \text{LastIn} \) is set to \( y \), so the consequence also holds.

\( \square \)

Invariant 4.5. If \( x \in \{6..12\} \) and \( y \in \{17\} \), then \( \text{rival}_y = x \).

Proof. While \( x \in \{6..12\} \) and \( y \in \{17\} \), the value of \( \text{rival}_y \) remains unchanged. The antecedent becomes true by event \( x.4 \) when \( \text{Want}[1 - \text{side}_x] \neq \text{NIL} \) and \( y \in \{17\} \), or by event \( y.15 \) when \( \text{LastIn} \neq y \) and \( x \in \{6..12\} \).

In the first case, when process \( x \) performs line 4, process \( x \) reads \( \text{Want}[1 - \text{side}_x] = \text{Want}[\text{side}_y] \) in \( x.4 \). Since process \( y \) is at line 17 at this time, there is no process \( r \) such that \( r \in \{2..14\} \) and \( \text{side}_r = \text{side}_y \) by the induction hypothesis. Thus, \( \text{Want}[\text{side}_y] = \text{NIL} \) by Observation 4.1. Hence, it is impossible for process \( x \) to perform line 4 with \( \text{Want}[1 - \text{side}_x] \neq \text{NIL} \) while process \( y \in \{17\} \).

In the second case, when process \( y \) performs line 15, \( \text{LastIn} = x \) by Observation 4.2. Hence, process \( y \) writes \( x \) to \( \text{rival}_y \) in \( y.15 \). Thus, \( \text{rival}_y = x \).

\( \square \)

Invariant 4.6. If \( x \in \{4\} \) and \( y \in \{17\} \), then \( \text{rival}_y = x \), \( \text{Spin}[x] = \text{RED} \), or \( \text{LastIn} \neq x \).
Proof. Consider a configuration in which \( x@4 \) and \( y@17 \). The most recent event was either \( x.3 \) or \( y.15 \). In \( x.3 \), process \( x \) sets \( Spin[x] \) to RED, so the consequence holds. In \( y.15 \), process \( y \) reads \( LastIn \) and writes it to \( rival_y \). Hence, just after \( y.15 \), either \( rival_y = x \) or \( LastIn \neq x \).

Invariants 4.5 and 4.6 imply the following invariant.

**Invariant 4.7.** If \( x@\{4..12\} \) and there is no process \( r \) other than \( x \) that has finished the doorway but not subsequently returned to the remainder section (i.e., \( r@\{3..17\} \) implies \( r = x \)), then \( Spin[x] \neq YELLOW \) or \( LastIn \neq x \).

Proof. Note that, when \( x \) performs line 4, it sets \( rival_x \) to the value of \( Want[1-side_x] \). Since \( side_x \) is a constant, only \( x \) can change \( Want[side_x] \), and \( x \) can only set \( Want[side_x] \) to \( x \) or NIL, \( Want[1-side_x] \) cannot be \( x \). Hence, when \( x@\{4..12\} \), \( rival_x \neq x \), so \( x \) does not access \( Spin[x] \) in line 8. \( LastIn \) is changed only when line 2 is performed. Therefore, while the antecedent remains true, the values of \( Spin[x] \) and \( LastIn \) remain unchanged.

The antecedent becomes true when process \( x \) performs line 3 or some process \( y \neq x \) finishes either the exit or abort protocol. In the first case, \( Spin[x] = RED \) just after \( x \) performs line 3. Thus, if \( x \) performs line 3 when there is no process \( r \neq x \) such that \( r@\{3..17\} \), then the consequence holds.

In the second case, either \( y \) performs line 17 or \( y \) performs line 15 with \( LastIn = y \) followed by line 16. If \( y \) performs line 15 with \( LastIn = y \) while process \( x@\{4..12\} \), then \( LastIn \neq x \), so the consequence holds. If \( y \) performs line 17 while process \( x@\{4..12\} \), then \( Spin[rival_y] \) is set to GREEN. By Invariant 4.5, \( rival_y = x \) when \( x@\{6..12\} \) and \( y@17 \). Also, by Invariant 4.6, \( rival_y = x \), \( Spin[x] = RED \), or \( LastIn \neq x \) when \( x@4 \) and \( y@17 \). Hence, just after \( y.17 \), \( Spin[x] = GREEN \) or RED, or \( LastIn \neq x \), so the consequence also holds.

The following invariant implies mutual exclusion.
Invariant 4.8. If x is in the critical section and y has finished the doorway but has not performed line 14 (i.e., y@{3..14}), then LastIn = y.

Proof. While x is in the critical section and y@{3..14}, the value of LastIn remains unchanged. If y finishes the doorway while x is in the critical section, then it sets LastIn = y. Process x can enter the critical section from lines 5, 6, 11, or 12. If x enters the critical section while y@{3..14}, then either rivalx = NIL, LastIn ≠ x, or Spin[x] = GREEN. If rivalx = NIL, then LastIn = y by Invariant 4.3. If LastIn ≠ x, then LastIn = y by Observation 4.2. If Spin[x] = GREEN, then LastIn = y by Invariant 4.4.

Lemma 4.9. The algorithm satisfies Mutual Exclusion.

Proof. Suppose, for contradiction, that two different processes p and q are in the critical section at the same time. By Invariant 4.8, LastIn = p and LastIn = q, which is a contradiction.

In the remainder of this section, we prove lockout freedom. We begin by presenting some properties of busy-waiting processes. Note that process p is busy-waiting when p@9 or p@12.

Invariant 4.10. If x@{7..12} and y@{9..12}, then Spin[x] ≠ RED or LastIn = x.

Proof. Spin[x] is set to RED only by x.3. Hence, while x@{7..12}, Spin[x] cannot be set back to RED. LastIn can be updated only by line 2. Hence, its value does not change while x@{7..12} and y@{9..12}.

If x performed line 6 while y@{9..12}, then either x entered the critical section or LastIn = x. Hence, either the antecedent is falsified or the consequence holds.

If y finished or skipped its line 8 while x@{7..12}, then by Invariant 4.3 with x and y interchanged, rivaly = x or LastIn = x when y performed line 7 or 8. If LastIn = x, then the consequence holds. If rivaly = x, then either y read Spin[x] ≠ RED in line 7 or
y wrote YELLOW to $Spin[x]$ in line 8. In both cases, $Spin[x] \neq RED$, so the consequence holds.

**Invariant 4.11.** If $x \in \{11..12\}$ and there is no process $r$ other than $x$ such that $r$ has finished the doorway but not subsequently returned to the remainder section (i.e., $r \in \{3..17\}$ implies $r = x$), then $Spin[x] = GREEN$ or $LastIn \neq x$.

*Proof.* While the antecedent holds, the values of $Spin[x]$ and $LastIn$ remain unchanged. If $x$ finishes line 9, then $x$ read $Spin[x] \neq RED$. Furthermore, if there is no process $y$ other than $x$ such that $y \in \{3..17\}$, then, by Invariant 4.7, $Spin[x] \neq YELLOW$ or $LastIn \neq x$. Therefore, $Spin[x] = GREEN$ or $LastIn \neq x$.

If some other process $y$ performs line 15 with $LastIn = y$ and skips line 17 while $x \in \{11..12\}$, then $LastIn \neq x$, so the consequence holds. If $y$ performs line 17 while $x \in \{11..12\}$, then by Invariant 4.5, $rival_y = x$ and $y$ sets $Spin[x]$ to GREEN.

The following invariant says that a process is not locked out in line 12 if $LastIn$ has been updated by another process.

**Invariant 4.12.** If $x \in 12$, then $Spin[x] = GREEN$ or $LastIn = x$.

*Proof.* Suppose, for contradiction, that $x \in 12$, $Spin[x] \neq GREEN$ and $LastIn \neq x$. Before $x$ is at line 12, $x$ performed line 11 with $LastIn = x$. Since $LastIn \neq x$ by assumption, some process $r$ other than $x$ performed line 2 while $x \in 12$, i.e., $x.11 \prec r.2 \prec x.12$.

Since the algorithm is abortable, line 2 can be performed several times while $x \in 12$. Suppose process $s$ is the first process that performs line 2 while $x \in 12$, i.e., $x.11 \prec s.2 \prec x.12$ and no process performs line 2 between $x.11$ and $s.2$. When $s \in 2$, all processes other than $x$ and $s$ are in the remainder section. Since $LastIn = x$, Invariant 4.11 implies that $Spin[x] = GREEN$.

Only $x$ can set $Spin[x]$ to RED. Thus, while $x \in 12$, $Spin[x]$ cannot be changed back to RED. Therefore, $Spin[x]$ was changed to YELLOW by some event $r.8$ with $rival_r = x$.
and $s.2 \prec r.8$. When $r.7$ was last performed, $\text{Spin}[x] = \text{RED}$. However, just before $s.2$, $\text{Spin}[x] = \text{GREEN}$, and $\text{Spin}[x]$ cannot be changed back to RED. Thus, $r.7 \prec s.2 \prec r.8$.

This is impossible since $s$ and $r$ enter the trying protocol with the same value of $\text{side}$.

**Invariant 4.13.** If $x \oplus 9$, then $\text{Spin}[p] \neq \text{RED}$ or there exists a process $y \oplus \{2..17\}$ such that $y \neq x$.

**Proof.** Suppose, for contradiction, that $x \oplus 9$, $\text{Spin}[x] = \text{RED}$ and there is no process $y \neq x$ such that $y \oplus \{2..17\}$. Note that $\text{Spin}[x]$ is set to RED only in $x.3$. Hence, if some process sets $\text{Spin}[x]$ to YELLOW or GREEN after $x$ last performed line 3, then $\text{Spin}[x] \neq \text{RED}$ until $x$ re-invokes the trying protocol and performs 3 again. Therefore, while $x \oplus \{4..9\}$, no process sets $\text{Spin}[x]$ to YELLOW or GREEN.

Process $x$ is at line 9 because $x$ performed line 4 with $\text{Want}[1 - \text{side}_{x}] \neq \text{NIL}$ and line 6 with $\text{LastIn} = x$. Observation 4.1 implies that, when $x$ performs line 4, there exists a process $r \neq x$ such that $r \oplus \{2..14\}$ and $\text{side}_{r} = 1 - \text{side}_{x}$. Then $r$ performed or skipped line 17 while $x \oplus \{6..8\}$; otherwise, $r \oplus \{2..17\}$ when $x \oplus 9$, which is a contradiction.

If $r$ performed line 17, then Invariant 4.5 implies that $\text{rival}_{r} = x$ when $x \oplus \{6..8\}$ and $r \oplus 17$. Hence, $r$ sets $\text{Spin}[x]$ to GREEN in $r.17$, which is a contradiction.

Thus, $r$ skipped line 17. This is because $\text{LastIn} = r$ when $r$ last performed line 15. Hence, $x.2 \prec r.2$. Note that $r$ did not perform line 2 between $x.2$ and $x.6$ since $\text{LastIn} = x$ when $x$ performed line 6. Thus, $x.6 \prec r.2$. Then, when $r$ last performed line 4, $\text{Want}[1 - \text{side}_{r}] = \text{Want}[\text{side}_{x}] = x$, so $r$ performed line 6. When $r$ last performed line 6, $\text{LastIn} = r$, so $r$ performed line 7. Since $\text{Spin}[x] = \text{RED}$, $r$ sets $\text{Spin}[\text{rival}_{r}] = \text{Spin}[x]$ to YELLOW in $r.8$, which is a contradiction.

**Lemma 4.14.** The algorithm satisfies Lockout Freedom.

**Proof.** Suppose, for the sake of contradiction, that process $p$ is busy-waiting forever at line 9 or 12 without aborting. Hence, $\text{Spin}[p] = \text{RED}$ if $p \oplus 9$ or $\text{Spin}[p] \neq \text{GREEN}$ if $p \oplus 12$. 

When $p@9$, Invariant 4.13 implies that there exists a process $q@\{2..17\}$ such that $q \neq p$. Thus, $q$ is eventually stuck at line 9 or line 12. Since $Spin[p] = \text{RED}$, Invariant 4.10 implies $\text{LastIn} = p$. Invariant 4.10 also implies that $Spin[q] \neq \text{RED}$. Thus, $q$ is not stuck at line 9 when $p@9$. If $q$ is stuck at line 12 when $p@9$, then Invariant 4.12 implies that $Spin[q] = \text{GREEN}$, so $q$ is not stuck at line 12 when $p@9$. This is a contradiction.

When $p@12$, Invariant 4.12 implies that $\text{LastIn} = p$, since $Spin[p] \neq \text{GREEN}$. Then Invariant 4.11 implies that there exists a process $q \neq p$ such that $q@\{3..17\}$. Thus, $q$ is eventually stuck at line 9 or line 12. If $q$ is stuck at line 9 when $p@12$, then Invariant 4.10 implies $Spin[q] \neq \text{RED}$, so $q$ is not busy-waiting forever at line 9. If $q$ is stuck at line 12 when $p@12$, Invariant 4.12 implies $Spin[q] = \text{GREEN}$, so $q$ is not busy-waiting forever at line 12. This is a contradiction. \hfill \Box

**Theorem 1.** There exists a local-spin abortable mutual exclusion algorithm that uses only read and write operations, has $\Theta(\log N)$ RMA complexity, and satisfies mutual exclusion, lockout freedom, bounded exit, and bounded abort.

### 4.3 Complexity of this algorithm

As in Yang and Anderson’s algorithm, each process performs $\Theta(\log N)$ RMAs to enter the critical section, on both the CC and DSM models. If a process aborts, it performs $O(\log N)$ RMAs. Even if there is no abort, each process still performs $\Theta(\log N)$ RMAs to enter the critical section.

Any abortable mutual exclusion algorithm solves mutual exclusion. Thus, any lower bound proof for mutual exclusion also applies to abortable mutual exclusion. Attiya, et al. [AHW08] showed that any mutual exclusion algorithm that uses only read, write, comparison-based operations and $\text{LL}/\text{SC}$ must have $\Omega(\log N)$ RMA complexity. Therefore, our algorithm is optimal in terms of RMA complexity.

The space complexity of our algorithm is $\Theta(N \log N)$, which is the same as Yang
and Anderson’s algorithm. As described in Section 3.1.1, Kim and Anderson modified Yang and Anderson’s algorithm to achieve $\Theta(N)$ space complexity. We believe we can modify Kim and Anderson’s algorithm in the same way we modified Yang and Anderson’s algorithm to get an abortable mutual exclusion algorithm with $\Theta(N)$ space complexity.

4.4 Applications of this algorithm

Although this algorithm is a simple modification of Yang and Anderson’s algorithm, there are several significant applications of this algorithm. Specifically, it can be used to obtain local-spin algorithms using only registers for FCFS abortable mutual exclusion, group exclusion and $k$-exclusion.

Danek [Dan11, DL08], presented a transformation from any local-spin abortable mutual exclusion algorithm to an FCFS local-spin abortable mutual exclusion algorithm using only registers. In this transformation, each process performs $\Theta(N)$ additional RMAs. Applying this transformation to our local-spin abortable mutual exclusion algorithm gives the first FCFS local-spin abortable mutual exclusion algorithm using only registers. The RMA complexity of the resulting algorithm is $\Theta(N)$.

Danek and Golab [DG08] presented a transformation from any local-spin mutual exclusion algorithm to an FCFS local-spin mutual exclusion algorithm that performs only $\Theta(\log N)$ additional RMAs. However, their transformation applied to our algorithm is not abortable. It is open whether there exists an FCFS local-spin abortable mutual exclusion algorithm using $o(N)$ RMAs.

Danek and Hadzilacos [DH04] presented a transformation from any FCFS local-spin abortable mutual exclusion algorithm to a group mutual exclusion algorithm using $\Theta(N)$ additional RMAs. Applying this transformation to our algorithm gives the first group mutual exclusion algorithm using only registers. The resulting group mutual exclusion algorithm has $\Theta(N)$ RMA complexity.
Any local-spin abortable mutual exclusion algorithm can also be transformed to a local-spin $k$-exclusion algorithm. The transformation uses $k$ instances of a local-spin abortable mutual exclusion algorithm. For $i \in \{1, \ldots, k\}$, let $\text{MutexTrying}_i()$, $\text{MutexCS}_i$, $\text{MutexExit}_i()$, and $\text{MutexAbort}_i()$ be the trying protocol, the critical section, the exit protocol, and the abort protocol of the $i$th instance.

When a process enters the trying protocol of the $k$-exclusion algorithm, it performs all $k$ instances of the abortable mutual exclusion algorithm concurrently (for example, repeatedly performing one step of each in round-robin order) until it enters the critical section of one of the instances.

When the process enters $\text{MutexCS}_j$, it finishes or aborts its execution of all other instances before entering the critical section of the $k$-exclusion algorithm. Note that a process can abort only during a busy-waiting period. Thus, if the process is not busy-waiting in $\text{MutexTrying}_i()$ for some $i \neq j$, then it continues performing $\text{MutexTrying}_i()$ until it enters a busy-waiting period or $\text{MutexCS}_i$. If it enters $\text{MutexCS}_i$ without busy-waiting, then it immediately performs $\text{MutexExit}_i()$. If it enters a busy-waiting period, then it immediately performs $\text{MutexAbort}_i()$. Note that, in any case, the process performs a bounded number of its own steps. When the process finishes the critical section of the $k$-exclusion algorithm, it performs $\text{MutexExit}_j()$.

Let $A'$ be the algorithm resulting from applying this transformation to some abortable mutual exclusion algorithm $A$. At any time, there is at most one process in the critical section of each instance of $A$, so there are at most $k$ processes in the critical section of $A'$. Hence $A'$ satisfies $k$-exclusion.

The only busy-waiting periods in $A'$ are in $\text{MutexTrying}_i()$ for all $i \in \{1, \ldots, k\}$. Since $\text{MutexTrying}_i()$ satisfies lockout freedom, no process is stuck forever in $\text{MutexTrying}_i()$. Hence, any process in the trying protocol eventually enters $\text{MutexCS}_i$ for some $i$, i.e., it eventually enters the critical section of $A'$. Thus, $A'$ satisfies lockout freedom.
Chapter 4. An Algorithm Using Only Registers

A process enters the critical section of $A'$, only if it has entered $\text{MutexCS}_j$ for some $j \in \{1, \ldots, k\}$ and has finished $\text{MutexABORT}_i()$ or $\text{MutexEXIT}_i()$ for all $i \neq j$. Thus, if at most $k - 1$ processes are in the critical section, then there exists some $\ell \in \{1, \ldots, k\}$ such that $\text{MutexCS}_\ell$ is available. Hence, some process in the trying protocol can eventually enter the critical section. Therefore, $A'$ satisfies $k$-deadlock freedom. Hence $A'$ is a $k$-exclusion algorithm.

If the RMA complexity of $A$ is $O(T)$, then the RMA complexity of $A'$ is $O(k \cdot T)$. Applying this transformation to our abortable mutual exclusion algorithm with $\Theta(\log N)$ RMA complexity gives a $k$-exclusion algorithm with $\Theta(k \cdot \log N)$ RMA complexity. This is the first local-spin $k$-exclusion algorithm using only read and write operations.

**Theorem 2.** There exists a local-spin $k$-exclusion algorithm with $\Theta(k \cdot \log N)$ RMA complexity that uses only read and write operations, and satisfies $k$-exclusion, lockout freedom, bounded exit, and $k$-deadlock freedom.

The $k$-exclusion algorithm obtained from this transformation does not tolerate any crash failures during the trying protocol or exit protocol. Suppose that a process $p$ enters $\text{MutexCS}_j$ and crashes before performing $\text{MutexABORT}_i()$ or $\text{MutexEXIT}_i()$ for all $i \neq j$. Hence, $p$ crashes during $\text{MutexTRYING}_i()$ for all $i \neq j$. Since the abortable mutual exclusion algorithm does not tolerate a crash failure, $p$'s crash during $\text{MutexTRYING}_i()$ may prevent other processes from entering $\text{MutexCS}_j$. Thus, even though only one process crashes, no process can enter the critical section of the $k$-exclusion algorithm.
Chapter 5

Bounded Space Algorithm 1

In this chapter, we present the first local-spin abortable mutual exclusion algorithm with bounded space, in which each process performs only $O(1)$ RMAs when no abort occurs. However, this algorithm is local-spin only in the CC model with a cache-update protocol.

5.1 Algorithm

Our local-spin abortable mutual exclusion algorithm uses `fetch_and_store`, `compare_and_swap`, and registers. The basic structure is similar to Craig’s mutual exclusion algorithm on the CC model, which is described in Section 3.1.2. In Craig’s algorithm, trying processes form a queue using one `fetch&store` object, Tail. If process $p$ becomes process $q$’s successor in the queue, then $p$ waits in the trying protocol for the shared variable $Lock[q]$ to be changed from LOCKED to UNLOCKED. Process $q$ changes $Lock[q]$ from LOCKED to UNLOCKED in its exit protocol, and $p$ can enter the critical section after reading $Lock[q] = UNLOCKED$.

In our algorithm, spin variable $Lock[q]$ is also used for process $q$ to inform its successor whether $q$ has aborted. When $q$ aborts, $q$ writes the ID of its predecessor to $Lock[q]$. When $p$ reads a process ID, $r$, from $Lock[q]$, $p$ knows that $q$ has aborted and $r$ was the predecessor of $q$. Hence, $r$ becomes $p$’s new predecessor, and $p$ waits for $Lock[r]$ instead
shared variables:

- Tail: ∈ \((\mathcal{P} \times \{0, 1\}) \cup \{\text{UNLOCKED}\}\), initially UNLOCKED (\(\mathcal{P}\) is a set of all processes)
- Lock\([p,\ face]\): ∈ \((\mathcal{P} \times \{0, 1\}) \cup \{\text{UNLOCKED, LOCKED}\}\), initially LOCKED

private variable:

- face ∈ \{0, 1\}, initially 0
- pred, initially UNLOCKED
- mylock

TryingProtocol()

1. if \(\neg \text{COMPARE\_AND\_SWAP}(\text{Lock}\[p,\ face\],\ pred,\ \text{LOCKED})\) then
2. \(\text{pred} := \text{FETCH\_AND\_STORE}(\text{Tail}, (p,\ face))\)
3. end if
4. if \(\text{pred} \neq \text{UNLOCKED}\) then
5. \(\text{mylock} := \text{Lock}[\text{pred}]\)
6. while \(\text{mylock} \neq \text{UNLOCKED}\) do
7. if \(\text{mylock} = \text{LOCKED}\) then
8. (If \(p\) wants to abort, then go to AbortProtocol().)
9. else if \(\text{COMPARE\_AND\_SWAP}(\text{Lock}[\text{pred}], \text{mylock}, \text{LOCKED})\) then
10. \(\text{pred} := \text{mylock}\)
11. end if
12. end while
13. \(\text{Lock}[\text{pred}] := \text{LOCKED}\)
14. end if

ExitProtocol()

15. \(\text{Lock}[p,\ face] := \text{UNLOCKED}\)
16. \(\text{face} := 1 - \text{face}\)

AbortProtocol()

17. \(\text{Lock}[p,\ face] := \text{pred}\)

of Lock\([q]\). After \(p\) reads \(\text{Lock}[q] \neq \text{LOCKED}\), \(p\) does not continue to spin on \(\text{Lock}[q]\). If \(\text{Lock}[q] = \text{UNLOCKED}\), then \(p\) enters the critical section. If \(\text{Lock}[q] = r\), then \(p\) spins on \(\text{Lock}[r]\). In order to allow \(\text{Lock}[q]\) to be reused, \(p\) resets \(\text{Lock}[q]\) to \text{LOCKED} after it reads \(\text{Lock}[q] \neq \text{LOCKED}\).

When a process \(q\) aborts during busy-waiting and re-enters the trying protocol, it checks whether its associated spin variable \(\text{Lock}[q]\) was accessed by another process since \(q\) last aborted. If so, then \(q\) is enqueued at the end of the queue. If not, then \(q\) reclaims its previous position in the queue. In this case, instead of performing FETCH\_AND\_STORE
on Tail, q resets Lock[q] to LOCKED and continues its trying protocol.

To avoid a race condition between the aborting process q and its successor p, both p and q try to reset Lock[q] to LOCKED using a compare_and_swap. If p’s cas succeeds, then q’s cas fails. In this case, q knows that p has accessed Lock[q] since q last aborted, so q does not reclaim the same position in the queue. If q’s cas succeeds, then p’s cas fails. In this case, p knows that q has re-entered the trying protocol and reclaimed the same position in the queue, so p keeps busy-waiting on Lock[q].

However, there is still a problem. If a process q finishes the exit protocol and enters the trying protocol again, then q cannot reuse Lock[q] until its previous successor resets Lock[q]. We can avoid this problem by assigning two different identities to each process. Each process, q, has a private Boolean variable, face, and two shared variables, Lock[q, 0] and Lock[q, 1], instead of Lock[q]. When a process performs fetch_and_store on Tail, it writes a pair, its ID and its face, to Tail, instead of only its ID.

A process toggles its face only when it finishes the critical section. If a process aborts and re-enters the trying protocol, it does not toggle its face. This is because, when q aborts and re-enters the trying protocol, Lock[q] is reset either by q or q’s previous successor. Figure 6 describes this algorithm in detail.

5.2 Proof of correctness

In this algorithm, when a process performs line 2, it enqueues itself with its face. However, a process performs line 2 only when the cas operation in line 1 fails. Hence, some passages do not perform line 2. This happens when a process aborted and then re-invokes the trying protocol before other processes notice the abort. In this case, this process reclaims the same position in the queue. Therefore, processes do not enter the critical section in the same order as they invoke the trying protocol. Instead, they enter the critical section in the order when they last perform line 2.
To capture this, we define an attempt, which is a part of an execution that starts when a process performs line 2. All attempts can be ordered according to when they start. Each attempt is identified with a process ID and its face, which are written to Tail when the attempt begins. The process ID and face of an attempt are used to index a shared array Lock that records the state of attempts. Other processes may continue to access Lock\[p, f\] to determine the state of an attempt by process \(p\) with face \(f\), after process \(p\) has invoked a new attempt with face \(1 - f\). We say that an attempt ends when the state information about it that is recorded in Lock is no longer available, which is when the process next performs line 2 with the same value of face. The following definitions formalize this.

**Definition 1.** An attempt \(A\) by process \(p\) is a part of an execution which begins when process \(p\) performs line 2, and ends when process \(p\) next performs line 2 with the same value of face. An attempt is live in any configuration between when it begins and ends. If a process \(p\) starts an attempt \(A\) by performing line 2 with face \(p = f\), then we say that the identity of attempt \(A\) is process \(p\) with face \(f\) and denote it by id\((A) = (p, f)\).

**Definition 2.** We say that attempt \(A\) precedes \(B\) if \(A\) begins before \(B\). We denote this by \(A \prec B\). Attempt \(A\) is the predecessor of attempt \(B\) if \(A \prec B\) and there is no attempt \(C\) such that \(A \prec C \prec B\). If attempt \(A\) is live in configuration \(C\) and id\((A) = (p, f)\), then we say that \(A\) is the current attempt by \((p, f)\) in \(C\).

In any configuration, it is impossible that two different live attempts have the same identity. However, there could be two live attempts \(A\) and \(B\) such that id\((A) = (p, 0)\) and id\((B) = (p, 1)\). Thus, there are at most \(2N\) live attempts in any configuration. Note that an attempt by process \(p\) is still live when \(p\) next finishes the exit or abort protocol. Also, note that there exists a live attempt by \((p, 0)\) at any configuration after process \(p\) has performed line 2 at least once, and there exists a live attempt by \((p, 1)\) at any configuration after process \(p\) has performed line 2 at least twice.
We say that an attempt $I$ performs some event, if $id(I) = (p, f)$, $I$ is live, and process $p$ performs the event with $face_p = f$. Also, we say that an attempt $I$ is in some line, if $id(I) = (p, f)$, $I$ is live, and process $p$ is in the line with $face_p = f$. For example, we say that an attempt $I$ is in the critical section, if $id(I) = (p, f)$, $I$ is live, and process $p$ is in the critical section with $face_p = f$.

In this algorithm, a process only accesses its private variables in lines 3, 5, 6, 7, 9 and 13. Hence, these are not events and $p@3$, $p@5$, $p@6$, $p@7$, $p@9$ and $p@13$ are never true. When a process decides to abort in line 7 of the trying protocol, it changes its next event from line 8 to line 14. Hence, when $p@14$, $p$ is still in the trying protocol and about to perform the abort protocol. Thus, a process $p$ is in the trying protocol if $p@\{2, 4, 8, 10, 11, 14\}$. We say that a process $p$ is in the hallway if $p@\{4, 8, 10, 11, 14\}$.

After $p$ performs line 10, if $mylock_p = \text{UNLOCKED}$, then $p@11$. if $mylock_p = \text{LOCKED}$ and $p$ decides to abort, then $p@14$. if $mylock_p = \text{LOCKED}$ and $p$ does not want to abort, then $p@10$, so $p$ is local-spinning, and if $mylock_p \notin \{\text{LOCKED, UNLOCKED}\}$, then $p@8$.

Process $p$ is in the critical section when $p@12$. The exit protocol and the abort protocol consist of one event, so we do not say that a process is in the exit protocol or the abort protocol. When $p@1$, process $p$ is in the remainder section.

**Observation 5.1.** If $p$ is the first process that performs line 2, then $pred_p \in P \times \{0, 1\}$ after $p$ performs line 2 for the second time. Otherwise, $pred_p \in P \times \{0, 1\}$ after $p$ performs line 2 at least once.

*Proof.* The value of $pred_p$ is set in line 2 or 9. $Tail$ is updated only when a process performs line 2 (i.e., when a new attempt begins). Each time a process performs line 2, it updates the value of $Tail$ with the identity of the new attempt. When line 2 is performed for the first time, the initial value of $Tail$, UNLOCKED, is returned. Otherwise, the identity of its predecessor is returned from $Tail$.

In line 9, $pred_p$ is set to $mylock_p$. Note that, by lines 5 and 6, line 9 is performed
only when $mylock_p \notin \{\text{LOCKED, UNLOCKED}\}$. Hence, immediately after $p$ performs line 9, $\text{pred}_p$ is set to some value in $P \times \{0, 1\}$.

\textbf{Observation 5.2.} Only process $p$ can set $\text{Lock}[p, 0]$ and $\text{Lock}[p, 1]$ to a value other than LOCKED. It sets $\text{Lock}[p, \text{face}_p]$ to UNLOCKED in line 12 in the exit protocol, and to some value in $P \times \{0, 1\}$ in line 14 in the abort protocol. A process $q \neq p$ can set $\text{Lock}[p, 0]$ or $\text{Lock}[p, 1]$ to LOCKED in line 8 or 11.

\textit{Proof.} The value of $\text{Lock}[p, \text{face}_p]$ can be updated in lines 1, 8, 11, 12 and 14. In lines 1, 8 and 11, $\text{Lock}[p, \text{face}_p]$ can be set to LOCKED. $\text{Lock}[p, \text{face}_p]$ can be set to some value other than LOCKED only when line 12 or 14 is performed by process $p$. When $p$ performs line 12, $\text{Lock}[p, \text{face}_p]$ is set to UNLOCKED. When $p$ performs line 14, $\text{Lock}[p, \text{face}_p]$ is set to $\text{pred}_p$.

If $p$ is the first process that performs line 2 in the execution, then $p$ enters the critical section immediately afterwards. In this case, $p$ can perform line 14 only after $p$ performs line 2 for the second time. Hence, by Observation 5.1, $\text{pred}_p \in P \times \{0, 1\}$ when $p$ performs line 14. Thus, $\text{Lock}[p, \text{face}_p]$ is set to some value in $P \times \{0, 1\}$ in $p$’s abort protocol.

\textbf{Observation 5.3.} In any configuration, if $\text{Lock}[p, f] \in P \times \{0, 1\}$, then $p$ is in the remainder section and immediately prior to last entering the remainder section, $p$ performed the abort protocol with $\text{face}_p = f$.

\textit{Proof.} Since $\text{Lock}[p, f] \in P \times \{0, 1\}$, Observation 5.2 implies that process $p$ performed the abort protocol with $\text{face}_p = f$ immediately prior to last entering the remainder section, and $\text{Lock}[p, f]$ has not been updated since then. If $p$ re-invokes the trying protocol after it last performed the abort protocol, then the cas in line 1 succeeds, so $\text{Lock}[p, f]$ is changed to LOCKED, which is a contradiction. Hence, after $p$ last performed the abort protocol, $p$ remains in the remainder section.

\textbf{Observation 5.4.} If a process performs the exit protocol and re-enters the trying protocol, it begins a new attempt.
Proof. The first time $p$ performs line 1 with $\text{face}_p = f$, the value of $\text{Lock}[p, f]$ is its initial value, LOCKED. If this attempt is the first attempt of the execution, then $\text{pred}_p = \text{UNLOCKED}$. Otherwise, by Observation 5.1, $\text{pred}_p \in \mathcal{P} \times \{0, 1\}$. In either case, the cas in line 1 fails and $p$ performs line 2.

Now consider any subsequent time process $p$ performs line 1 with $\text{face}_p = f$. When it last performed the exit protocol, it set $\text{Lock}[p, 1 - f]$ to UNLOCKED on line 12 and set $\text{face}_p$ to $f$ on line 13. The previous time it performed the exit protocol, it set $\text{Lock}[p, f]$ to UNLOCKED and set $\text{face}_p$ to $1 - f$. Between these two times process $p$ can only change $\text{Lock}[p, f]$ to LOCKED. By Observation 5.2, other processes can only change $\text{Lock}[p, f]$ to LOCKED.

Thus $\text{Lock}[p, f]$ is either LOCKED or UNLOCKED when $p$ re-enters the trying protocol after it last performed the exit protocol. By Observation 5.1, $\text{pred}_p \in \mathcal{P} \times \{0, 1\}$. Thus, the cas in line 1 fails and $p$ performs line 2.

Therefore, in both cases, $p$ starts a new attempt after it last performed the exit protocol.

\[ \Box \]

Observation 5.5. Suppose that a process $p$ performs the abort protocol and re-enters the trying protocol. It begins a new attempt only if $\text{Lock}[p, \text{face}_p]$ was changed by another process between when $p$ last performed the abort protocol and when $p$ re-enters the trying protocol.

Proof. Suppose process $p$ re-enters the trying protocol after it last performed the abort protocol with $\text{face}_p = f$. If no process changes $\text{Lock}[p, f]$ after $p$ last performed the abort protocol and before $p$ re-enters the trying protocol, then $\text{Lock}[p, \text{face}_p] = \text{pred}_p$ when $p$ performs line 1. In this case, the cas succeeds, so $\text{Lock}[p, f]$ is set to LOCKED and $p$ skips line 2. If $\text{Lock}[p, f]$ was changed between when $p$ last performed the abort protocol and when $p$ re-enters the trying protocol, then, by Observation 5.2, $\text{Lock}[p, f] = \text{LOCKED}$. In this case, the cas fails, and $p$ will perform line 2 with $\text{face}_p = f$. Therefore, $p$ starts a new attempt. \[ \Box \]
After another process reads $\text{Lock}[p,f]$ to find the state of the attempt by $(p,f)$, it resets $\text{Lock}[p,f]$ to LOCKED. Then, process $p$ begins a new attempt when it re-invokes the trying protocol. Thus, between these two times, no process accesses $\text{Lock}[p,f]$ even though the attempt by $(p,f)$ is live. We say that such live attempt is *dormant* during that time. $\text{Lock}[p,f]$ can be accessed by other processes only when the attempt by $(p,f)$ is not dormant. The following definition formalizes this, and it is used to prove the correctness of the algorithm.

**Definition 3.** Suppose that an attempt $A$ with identity $(p,f)$ begins immediately before configuration $C'$. Then attempt $A$ is *dormant* in a configuration $C$ if

$A$ is live,

$\text{Lock}[p,f] = \text{LOCKED}$ in $C$, and

either $p$ performed the exit protocol with $\text{face}_p = f$ between $C'$ and $C$,

or $p@\{1,2\}$ in $C$ and, immediately prior to last entering the remainder section, $p$ performed the abort protocol with $\text{face}_p = f$.

Note that the exit protocol consists of a single step (line 12) followed by a local operation (line 13), and the abort protocol consists of a single step (line 14).

**Observation 5.6.** If process $p$ is in the hallway or the critical section with $\text{face}_p = f$, then the current attempt by $(p,f)$ is not dormant.

*Proof.* Suppose, for contradiction, that, in configuration $C$, $p$ is in the hallway or the critical section with $\text{face}_p = f$ and the current attempt $I$ by $(p,f)$ is dormant. Then $\text{Lock}[p,f] = \text{LOCKED}$ and $p$ performed the exit protocol with $\text{face}_p = f$ after $I$ began but before $C$. When $p$ last performed the exit protocol with $\text{face}_p = f$, it toggled $\text{face}_p$ from $f$ to $1-f$. Thus, after this step, but before $C$, $p$ toggled $\text{face}_p$ from $1-f$ to $f$ by performing the exit protocol with $\text{face}_p = 1-f$. However, by Observation 5.4, $p$ started a new attempt with $\text{face}_p = f$ when $p$ next re-entered the trying protocol. Thus, $I$ ends before $C$. This contradicts the fact that $I$ is dormant in $C$. □
Observation 5.7. Once an attempt becomes dormant, it remains dormant until it ends.

Proof. Suppose, for contradiction, that there exist two consecutive configurations \( C \) and \( \hat{C} \) such attempt \( A \) by \((p,f)\) is live and dormant in \( C \), and \( A \) is live but not dormant in \( \hat{C} \). Then \( \text{Lock}[p,f] = \text{LOCKED} \) in \( C \). Let \( C' \) be the configuration immediately before \( A \) began.

First suppose that \( p \) performed line 12 with \( \text{face}_p = f \) between \( C' \) and \( C \) and, hence, between \( C' \) and \( \hat{C} \). Then \( \text{Lock}[p,f] \neq \text{LOCKED} \) in \( \hat{C} \) since \( A \) is not dormant in \( \hat{C} \). Thus, between \( C \) and \( \hat{C} \), \( \text{Lock}[p,f] \) is changed from \( \text{LOCKED} \) to some other value. By Observation 5.2, this is because \( p \) performs line 12 or 14 with \( \text{face}_p = f \) between \( C \) and \( \hat{C} \). Hence, in \( C \), \( p \) is in the hallway or in the critical section with \( \text{face}_p = f \). By Observation 5.6, \( A \) is not dormant in \( C \), which is a contradiction.

Hence, \( p \oplus \{1,2\} \) in \( C \) and, immediately prior to last entering the remainder section, \( p \) performed the abort protocol with \( \text{face}_p = f \). By Observation 5.2, \( p \) set \( \text{Lock}[p,f] \) to some value in \( P \times \{0,1\} \) when it performed the abort protocol. Since \( \text{Lock}[p,f] = \text{LOCKED} \) in \( C \), Observation 5.5 implies that \( p \) begins a new attempt when \( p \) re-enters the trying protocol. Since \( A \) is not dormant in \( \hat{C} \), between \( C \) and \( \hat{C} \), either \( p \) performs line 2 or \( \text{Lock}[p,f] \) changes from \( \text{LOCKED} \) to some other value. If \( p \) performs line 2 between \( C \) and \( \hat{C} \), \( A \) ends and, hence, \( A \) is not live in \( \hat{C} \). Since \( p \oplus \{1,2\} \) in \( C \), Observation 5.2 implies that \( \text{Lock}[p,f] \) could not have changed from \( \text{LOCKED} \) to some other value between \( C \) and \( \hat{C} \). In both cases, we have a contradiction.

\[ \square \]

Observation 5.8. If \( p \oplus 2 \) and immediately prior to last entering the remainder section, \( p \) performed the abort protocol with \( \text{face}_p = f \), then \( \text{Lock}[p,f] = \text{LOCKED} \).

Proof. Since \( p \oplus 2 \), when \( p \) last performed line 1, the \text{CAS} failed. Since \( p \) aborted in its last passage, \( \text{Lock}[p,f] \) was last set by \( p \) in line 14 or by another process. In line 14, \( p \) sets \( \text{Lock}[p,f] \) to \( \text{pred} \), so if \( \text{Lock}[p,f] \) was last set by \( p \) in line 14, then the \text{CAS} in line 1 succeeds. Hence, \( \text{Lock}[p,f] \) was last set by another process. By Observation 5.2, \( \text{Lock}[p,f] \) was last set to \( \text{LOCKED} \). Thus, \( \text{Lock}[p,f] = \text{LOCKED} \).  

\[ \square \]
Observation 5.9. If an attempt $A$ with identity $(p, f)$ is live, but not dormant in configuration $C'$ and becomes dormant in the next configuration $C$, then there exists an attempt $B \neq A$ that set $\text{Lock}[p, f]$ to LOCKED between $C'$ and $C$. Also, if $p$ performed the abort protocol after $A$ began, but before $C'$, then $A$ is in the remainder section in $C'$.

Proof. Suppose that $A$ is not dormant in $C'$ and becomes dormant in $C$. Then, between $C'$ and $C$, $\text{Lock}[p, f]$ was changed to LOCKED, $p$ performed the exit protocol with $\text{face}_p = f$, or immediately prior to last entering the remainder section, $p$ performed the abort protocol with $\text{face}_p = f$. Note that, immediately after $p$ performs the exit or abort protocol with $\text{face}_p = f$, $\text{Lock}[p, f] \in \{\text{UNLOCKED}\} \cup \mathcal{P} \times \{0, 1\}$, so $A$ is not dormant. Therefore, $\text{Lock}[p, f]$ was changed to LOCKED between $C'$ and $C$. Since $A$ is dormant in $C$, either $p$ performed the exit protocol with $\text{face}_p = f$ after $A$ began and before $C'$, or $p@\{1, 2\}$ in $C$ and $p$ performed the abort protocol with $\text{face}_p = f$ after $A$ began and before $C'$.

Suppose $p$ performed the exit protocol with $\text{face}_p = f$ after $A$ began and before $C'$. Then, immediately afterwards, $\text{face}_p = 1 - f$. If $A$ changed $\text{Lock}[p, f]$ between $C'$ and $C$, then, in $C'$, $\text{face}_p = f$. Hence, between these steps, $p$ performed the exit protocol with $\text{face}_p = 1 - f$. By Observation 5.4, $p$ began a new attempt when it subsequently re-entered the trying protocol. Thus, $A$ is not live in $C'$, which is a contradiction. Therefore, in this case, some other attempt set $\text{Lock}[p, f]$ to LOCKED between $C'$ and $C$.

Otherwise, $p@\{1, 2\}$ in $C$, and immediately prior to last entering the remainder section, $p$ performed the abort protocol with $\text{face}_p = f$ after $A$ began and before $C'$. Note that, in $C'$, if $p@2$, then by Observation 5.8, $\text{Lock}[p, f] = \text{LOCKED}$. Thus, $A$ is dormant in $C'$, which is a contradiction. Recall that $p$ does not perform the exit or abort protocol between $C'$ and $C$. Hence, in $C'$, $p$ is in the remainder section, i.e., $p@1$.

Suppose, for contradiction, that $A$ set $\text{Lock}[p, f]$ to LOCKED between $C'$ and $C$. Then, since $p@1$ in $C'$, $p$ performed a successful cas on line 1 and skipped line 2. Hence, so $p@4$ in $C$, which is a contradiction. Thus, in this case, some other attempt set
Lemma 5.10. Consider any configuration $C$. Let $I$ be the current attempt by $(p, \text{face}_p)$ in $C$.

1. If process $p$ is in the critical section, then all attempts $J \prec I$ are either dormant or not live.

2. If $I$ is not dormant and $\text{pred}_p = (q, g)$, then the current attempt $H$ by $(q, g)$ is not dormant, $H \prec I$, and all attempts $J$, where $H \prec J \prec I$, have aborted and are either dormant or not live.

3. If $I$ is not dormant and $\text{pred}_p = (q, g)$, then for every other process $r$ such that $\text{pred}_r = (q, g)$, the current attempt by $(r, \text{face}_r)$ in $C$ is dormant.

4. If $p @ 8$, $\text{pred}_p = (r, h)$, $\text{mylock}_p = \text{Lock}[r, h] = (q, g)$, then $(r, h) \neq (q, g)$ and $H \prec G \prec I$, where $G$ is the current attempt by $(r, h)$ and $H$ is the current attempt by $(q, g)$. Also, $H$ is not dormant, $G$ is in the remainder section, immediately prior to last entering the remainder section, $G$ aborted, and all attempts $J$, where $H \prec J \prec G$ or $G \prec J \prec I$, have aborted and are either dormant or not live.

Proof. Suppose there is an execution that contains a configuration in which one or more of these claims does not hold. Let $C'$ be the first such configuration in the execution, and let $f = \text{face}_p$ in $C$.

Case 1: (1) is violated in $C$. Then, in $C$, process $p$ is in the critical section and some attempt $J \prec I$ is live and not dormant. Therefore, $I$ is not the first attempt of the execution. By Observation 5.1, $\text{pred}_p \in P \times \{0, 1\}$ and, hence, $p$ entered the critical section from line 11. When $p$ last performed line 5, $\text{mylock}_p = \text{UNLOCKED}$. Hence, when $p$ last performed line 4 or 10, $\text{Lock}[\text{pred}_p] = \text{UNLOCKED}$.

Let $C'$ be the configuration immediately before $p$ last performs line 4 or 10 prior to $C$. Then, between $C'$ and $C$, $p$ did not perform line 2 and the value of $\text{face}_p$ has not changed. Thus, $I$ is live in $C'$. Let $(q, g) = \text{pred}_p$ in $C'$ and let $H$ be the current attempt
by \((q,g)\) in \(C'\). Since (2) holds in \(C'\), \(H \prec I\) and all attempts \(J\), where \(H \prec J \prec I\), have aborted and are either dormant or not live in \(C'\). Hence, by Observation 5.7, they are also dormant or not live in \(C\). Note that, in \(C'\), \(\text{Lock}[q,g] = \text{Lock}[\text{pred}_p] = \text{UNLOCKED}\). Hence, by Observation 5.2, process \(q\) performed line 12 with \(\text{face}_q = g\) before \(C'\), and \(\text{Lock}[q,g]\) has not changed since then.

Let \(C''\) be the configuration immediately before \(q\) last performed line 12 with \(\text{face}_q = g\) prior to \(C'\). Note that, in \(C''\), \(q\) was in the critical section and \(\text{face}_q = g\). Let \(H'\) be the current attempt by \((q,g)\) in \(C''\). If \(H''\) ended between \(C''\) and \(C'\), then \(q\) performed the exit protocol with \(\text{face}_q = 1 - g\) between \(C''\) and \(C'\). Then, immediately before this step, the current attempt \(H''\) by \((q,1 - g)\) was in the critical section. Then \(H' \prec H''\) and, by (1), \(H'\) was dormant when \(H''\) was in the critical section. Thus, \(\text{Lock}[q,g]\) was set to LOCKED after \(C''\), which is a contradiction. Hence, \(H' = H\) and \(H\) is live in \(C''\).

Since (1) holds in \(C''\), all attempts \(J \prec H\) are either dormant or not live in \(C''\). Thus, by Observation 5.7, they are also dormant or not live in \(C\). Hence, \(H\) is live and not dormant in \(C\).

Note that \(q\) performed line 12 with \(\text{face}_q = g\) between \(C''\) and \(C'\) and \(p\) set \(\text{Lock}[q,g]\) to LOCKED in line 11 between \(C'\) and \(C\). Hence, \(H\) is either dormant or not live in \(C\). This is a contradiction.

**Case 2:** (2) is violated in \(C\). Then, in \(C\), \(I\) is not dormant, \(\text{pred}_p = (q,g)\), and either the current attempt \(H\) by \((q,g)\) is dormant, \(H \not\prec I\), or there exists an attempt \(J\), where \(H \prec J \prec I\), such that \(J\) has not aborted, or \(J\) is live and not dormant. Let \(C'\) be the configuration immediately before \(C\).

Suppose that \(p\) performed line 2 with \(\text{face}_p = f\) between \(C'\) and \(C\). This begins \(I\), so \(I\) is not live in \(C'\). Since \(\text{pred}_p\) was set to \((q,g)\) by line 2, \(H\) is the predecessor of \(I\) and \(H\) was the last attempt started before \(C'\). It follows that \(H\) is dormant in \(C\). When \(H\) begins, it is not dormant. Let \(D'\) be the last configuration in which \(H\) is live but not dormant and let \(D\) be the next configuration. Hence, \(H\) is dormant in \(D\) and \(D'\) occurs
before $C$. By Observation 5.9, there exists an attempt $B \neq H$ that set $\text{Lock}[q,g]$ to LOCKED between $D'$ and $D$. Let $(s,t)$ be the identity of $B$. Then $s$ performed line 8 or 11 with $\text{pred}_s = (q,g)$ between $D'$ and $D$. By Observation 5.6, $B$ is not dormant in $D'$. Since (2) holds in $D'$, $H \prec B$. This contradicts that $H$ is the last attempt that began before $C'$. Therefore, $p$ does not perform line 2 with $\text{face}_p = f$ between $C'$ and $C$. Since $I$ is live in $C$, $I$ is live in $C'$.

Thus, by Observation 5.7, $I$ is not dormant in $C'$. Next suppose that $p$ performed lines 8 and 9 between $C'$ and $C$. As a result, the value of $\text{pred}_p$ is updated to $(q,g)$. Let $(r,h) = \text{pred}_p$ in $C'$ and let $G$ be the current attempt by $(r,h)$ in $C'$ and $C$. Then, in $C'$, $p@8$ and $\text{Lock}[r,h] = \text{mylock}_p = (q,g)$. In $C'$, since (4) holds, $(r,h) \neq (q,g)$, $H \prec G \prec I$, $H$ is not dormant, $G$ has aborted and is in the remainder section, and all attempts $J$, where $H \prec J \prec G$ or $G \prec J \prec I$, have aborted and are either dormant or not live in $C'$. Note that $\text{Lock}[r,h]$ becomes LOCKED between $C'$ and $C$ since the $\text{cas}$ in line 8 succeeds. Hence, in $C$, $G$ becomes dormant. Therefore, all attempts $J$, where $H \prec J \prec I$, have aborted and are either dormant or not live in $C$. If follows that $H$ is dormant in $C$. However, $H$ is not dormant in $C'$. Between $C'$ and $C$, since $\text{Lock}[q,g]$ is not changed and $q$ performs no steps, $H$ is not dormant in $C$. Therefore, we have a contradiction. Hence, $p$ does not perform lines 8 and 9 between $C'$ and $C$. Thus, $\text{pred}_p$ does not change between $C'$ and $C$ and, therefore, $\text{pred}_p = (q,g)$ in $C'$.

By (2), $H \prec I$, $H$ is not dormant in $C'$. Also, all attempts $J$, where $H \prec J \prec I$, have aborted and are either dormant and not live in $C'$ and, so, in $C$. It follows that $H$ is dormant in $C$. (3) implies that, for every process $s \neq p$ such that $\text{pred}_s = (q,g)$, the current attempt by $(s,\text{face}_s)$ is dormant in $C'$. Since $p$ did not perform lines 8 and 9 between $C'$ and $C$, by Observation 5.6, no such process $s$ is in the hallway, so cannot perform line 8 or 11 between $C'$ and $C$. Hence $H$ is not dormant in $C$. This is a contradiction.

Case 3: (3) is violated in $C$. Then, in $C$, $I$ is not dormant, $\text{pred}_p = (q,g)$, and there
exists a process \( r \neq p \) such that \( \text{pred}_r = (q, g) \) and the current attempt \( G \) by \( (r, \text{face}_r) \) in \( C \) is not dormant.

Let \( C' \) be the configuration immediately before \( C \), and let \( H \) be the current attempt by \( (q, g) \) in \( C \). Since (3) holds in \( C' \), between \( C' \) and \( C \), \( I \) or \( G \) began by performing line 2 with \( \text{face}_p = f \), or one of \( \text{pred}_p \) or \( \text{pred}_r \) is changed to \( (q, g) \) in line 9. First consider the case that \( I \) began between \( C' \) and \( C \). Then, \( H \) is live in \( C' \) and the predecessor of \( I \). Note that \( G \) is live in \( C' \) and, by Observation 5.7, \( G \) is not dormant in \( C' \). The value of \( \text{pred}_r \) is not changed between \( C' \) and \( C \), so \( \text{pred}_r = (q, g) \) in \( C' \). Since (2) holds in \( C' \), \( H \prec G \) and all attempts \( J \), where \( H \prec J \prec G \), have aborted and are either dormant or not live in \( C' \) and, hence, in \( C \). This contradicts the fact that \( H \) is the predecessor of \( I \), \( I \) is not dormant in \( C \), and \( I \neq G \). The case that \( G \) began between \( C' \) and \( C \) is symmetric.

Now consider the case that \( \text{pred}_p \) is changed to \( (q, g) \) in line 9 between \( C' \) and \( C \). Then, \( G, H \) and \( I \) are all live in \( C' \). Note that, in \( C' \), \( \text{pred}_r = (q, g) \) and, by Observation 5.7, \( G \) is not dormant. Since (2) holds in \( C' \), \( H \prec G \) and all attempts \( J \), where \( H \prec J \prec G \), have aborted and are either dormant or not live in \( C' \) and, so, in \( C \). Since \( I \) is live and not dormant in \( C' \), \( G \preceq I \). But \( \text{id}(G) \neq \text{id}(I) \), so \( G \neq I \). Thus \( H \prec G \prec I \).

Let \( (s, t) = \text{pred}_p \) in \( C' \), and let \( B \) be the current attempt by \( (s, t) \) in \( C' \) and \( C \). Note that, \( p@8 \) in \( C' \) and the cas on line 8 was successful, so \( \text{mylock}_p = \text{Lock}[s, t] = (q, g) \), and \( \text{Lock}[s, t] \) becomes LOCKED between \( C' \) and \( C \). Since (4) holds in \( C' \), \( (s, t) \neq (q, g) \), \( H \prec B \prec I \), and all attempts \( J \), where \( H \prec J \prec B \) or \( B \prec J \prec I \), have aborted and are either dormant or not live in \( C' \) and, hence, in \( C \). Since \( G \) is live and not dormant, and \( H \prec G \prec I \), it follows that \( G = B \). Hence, \( \text{pred}_p = (r, h) \) in \( C' \). Also by (4), \( G \) is in the remainder section in \( C' \) and, hence, in \( C \), and, immediately prior to last entering the remainder section, \( G \) aborted. Therefore, in \( C \), \( G \) becomes dormant, which contradicts the assumption that \( G \) is not dormant in \( C \). The case that \( \text{pred}_r \) is changed to \( (q, g) \) in line 9 between \( C' \) and \( C \) is symmetric.
Case 4: We will prove that (4) is not violated in $C$. Suppose that, in $C$, $p@8$, $\text{pred}_p = (r, h)$ and $\text{mylock}_p = \text{Lock}[r, h] = (q, g)$. Let $C'$ be the configuration immediately before $C$. We first show that $(r, h) \neq (q, g)$. Since $\text{Lock}[r, h] = (q, g)$ in $C$, by Observation 5.3, $G$ is in the remainder section and, immediately prior to last entering the remainder section, $G$ performed the abort protocol. Immediately before that, $G$ was in the hallway and $\text{pred}_r = (q, g)$. Thus, by Observation 5.6, $G$ was not dormant. Hence, by (2), the current attempt by $(q, g)$ precedes $G$. Hence, $(q, g) \neq (r, h)$.

We will prove the rest of the consequence is true by considering three cases, depending on whether $p$ performs the step and whether $\text{Lock}[r, h]$ changes between $C'$ and $C$. First suppose that $p$ performs the step between $C'$ and $C$. Since $p@8$ in $C$, $p$ performed line 4 or 10 between $C'$ and $C$, so $\text{pred}_p$ and $\text{Lock}[r, h]$ do not change. Thus, $\text{pred}_p = (r, h)$ and $\text{Lock}[r, h] = (q, g)$ in $C'$. Also, $I$, $G$ and $H$ are all live in $C'$. Since $p$ is in the hallway with $\text{face}_p = f$ in $C'$ and $C$, by Observation 5.6, $I$ is not dormant in $C'$ or $C$. Since (2) holds in $C'$ and $\text{pred}_p = (r, h)$ in $C'$, $G$ is not dormant in $C'$ and $G \prec I$. Also, all attempts $J$, where $G \prec J \prec I$, have aborted and are either dormant or not live in $C'$ and, hence, in $C$. Since $\text{Lock}[r, h] = (q, g)$ in $C'$, by Observation 5.3, $G$ is in the remainder section in $C'$ and immediately prior to last entering the remainder section, $G$ performed the abort protocol with $\text{face}_r = h$. Let $C''$ be the configuration immediately before $G$ last performed the abort protocol prior to $C''$. Since $G$ is live and not dormant in $C'$, $G$ is also live and not dormant in all configurations between $C''$ and $C'$, by Observation 5.7. Also, in $C''$, $\text{pred}_r = (q, g)$ since $\text{Lock}[r, h]$ was set to $(q, g)$ when $G$ performed the abort protocol. Let $H'$ be the current attempt by $(q, g)$ in $C''$. Since (2) holds in $C''$ and $\text{pred}_r = (q, g)$, $H'$ is not dormant in $C''$, $H' \prec G$, and all attempts $J$, where $H' \prec J \prec G$, have aborted and are either dormant or not live in $C''$ and, hence, in $C$.

Now we show that $H' = H$. Suppose, for contradiction, that $H' \neq H$. Recall that $H'$ is not dormant in $C''$ and $H$ is live in $C'$ and $C$. Hence, between $C''$ and $C'$, there exists a configuration $\hat{C}$ in which $H'@2$. 
If $H'$ performed the abort protocol immediately prior to last entering the remainder section, then by Observation 5.8, $\text{Lock}[q, g] = \text{LOCKED}$, so $H'$ is dormant in $\hat{C}$. Otherwise, $q$ performed the exit protocol with $\text{face}_q = 1 - g$ immediately prior to last entering the remainder section. Then, immediately before this step, the current attempt $H''$ by $(q, 1 - g)$ was in the critical section. Then $H' \prec H''$ and, by (1), $H'$ was dormant when $H''$ was in the critical section. Thus, by Observation 5.7, $H'$ is dormant in $\hat{C}$.

Therefore, there exist a pair of consecutive configurations $D'$ and $D$ between $C''$ and $\hat{C}$ such that $H'$ is live and not dormant in $D'$ and $H'$ is dormant in $D$. Recall that, between $C''$ and $C'$, $G$ is live and not dormant, $id(G) = (r, h)$, $\text{pred}_r = (q, g)$, and $G$ is in the remainder section. Hence, by (3), for every other process $s$ such that $\text{pred}_s = (q, g)$, the current attempt by $(s, \text{face}_s)$ is dormant between $C''$ and $C'$. Hence, Observation 5.6 implies that such a process $s$ cannot perform line 8 or 11 between $C''$ and $C'$. Thus, by Observation 5.2, no attempt can set $\text{Lock}[q, g]$ to LOCKED between $D'$ and $D$. Therefore, $H'$ is not dormant in $D$, which is a contradiction. Hence, $H' = H$.

Note that, since $H$ is live and not dormant in $C'$ and, $q$ does not perform the step and $\text{Lock}[q, g]$ is not changed between $C'$ and $C$, $H$ is also not dormant in $C$. Therefore, $H \prec G \prec I$, $H$ is not dormant in $C$, $G$ is in the remainder section in $C$, immediately prior to last entering the remainder section, $G$ aborted, and all attempts $J$, where $H \prec J \prec G$ or $G \prec J \prec I$, have aborted and are either dormant or not live. Thus, (4) holds in $C$.

Next consider the case that $p$ does not perform the step and $\text{Lock}[r, h]$ is not changed between $C'$ and $C$. Then $p@8$, $\text{pred}_p = (r, h)$ and $\text{mylock}_p = \text{Lock}[r, h] = (q, g)$ in $C'$. Then, by (4), $H \prec G \prec I$, $H$ is not dormant in $C'$, $G$ is in the remainder section in $C'$ and, immediately prior to last entering the remainder section, $G$ aborted. Also, all attempts $J$, where $H \prec J \prec G$ or $G \prec J \prec I$, have aborted and are either dormant or not live in $C'$ and, hence, in $C$. Since $\text{Lock}[r, h] = (q, g)$ in $C$, by Observation 5.3, $G$ is in the remainder section in $C$.

Suppose, for contradiction, that $H$ is dormant in $C$. Then by Observation 5.9, there
exists an attempt $B \neq H$ that sets $\text{Lock}[q,g]$ to LOCKED between $C'$ and $C$. Let $(s,t)$ be the identity of $B$. Then $s$ performed line 8 or 11 with $\text{pred}_s = (q,g)$ between $C'$ and $C$. Hence, by Observation 5.6, $B$ is not dormant in $C'$. By (2), $H \prec B$, and all attempts $J$, where $H \prec J \prec B$, have aborted and are either dormant or not live in $C'$. Since both $B$ and $G$ are live and not dormant in $C'$ and all attempts $J$, where $H \prec J \prec G$ or $H \prec J \prec B$, have aborted and are either dormant or not live in $C'$, $B = G$. However, in $C'$, $G$ is in the remainder section and $B$ is in the hallway, so $B \neq G$, which is a contradiction. Therefore, $H$ is not dormant in $C$. Hence (4) holds in $C$.

Finally, consider the case that $p$ does not perform the step and $\text{Lock}[r,h]$ is changed (to $(q,g)$) between $C'$ and $C$. Then $p@8$, $\text{pred}_p = (r,h)$ and $\text{mylock}_p = (q,g)$ in $C'$. By Observation 5.2, the current attempt $G$, by $(r,h)$, performed the abort protocol between $C'$ and $C$. Hence, in $C'$, $G$ is in the hallway and $\text{pred}_r = (q,g)$. By Observation 5.6, $G$ is not dormant in $C'$. Hence, by (2), $H \prec G$ and $H$ is not dormant in $C'$. Since $(r,h) \neq (q,g)$, $H$ is not dormant in $C$. Also, all attempts $J$, where $H \prec J \prec G$, have aborted and are either dormant or not live in $C'$, and hence, in $C$. Since $I$ is in the hallway in $C'$, by Observation 5.6, $I$ is not dormant in $C'$. Since $\text{pred}_p = (r,h)$ in $C'$, (2) implies that $G \prec I$ and $G$ is not dormant in $C'$. Also, all attempts $J$, where $G \prec J \prec I$, have aborted and are either dormant or not live in $C'$ and, hence, in $C$. Thus, $H \prec G \prec I$ and all attempts $J$, where $H \prec J \prec G$ or $G \prec J \prec I$, have aborted and are either dormant or not live in $C$. Hence, (4) holds in $C$.

The next result follows from part (1) of Lemma 5.10.

**Lemma 5.11.** The algorithm satisfies mutual exclusion.

The next result implies that the algorithm satisfies lockout freedom. The proof relies on the fact that, if a process aborts and re-enters the trying protocol, but no other process notices, then it is treated as if it was busy-waiting in the previous passage.
Lemma 5.12. In any execution, there is no attempt that aborts infinitely often or remains forever in the trying protocol without aborting.

Proof. Suppose, for contradiction, that there exists an execution in which some attempts either remain busy-waiting forever without aborting or abort infinitely often. Let $I$ be the earliest attempt among such attempts, and let $(p, f) = id(I)$. Note that $I$ is not the first attempt. This is because the initial value of $Tail$ is UNLOCKED, so if $I$ is the first attempt, then $I$ enters the critical section after performing lines 2 and 3.

Since $I$ remains busy-waiting forever or aborts infinitely often, $I$ performs line 4 or 10 infinitely often. By the definition of $I$, each attempt $J < I$ eventually ends or the process performing it remains in the remainder section. Consider a configuration $C$ in which this has occurred for every attempt $J < I$ and $p$ is about to perform line 4 or 10.

By Observation 5.1, $\text{pred}_p \in \mathcal{P} \times \{0, 1\}$ in $C$. Let $(q, g)$ be the value of $\text{pred}_p$ in $C$, and let $H$ be the current attempt by $(q, g)$ in $C$. By part (2) of Lemma 5.10, $H < I$ and $H$ is live and not dormant in $C$. Hence $\text{Lock}[q, g] \neq \text{LOCKED}$ in $C$ and $H$ remains in the remainder section after $C$. Also, since $I$ never enters the critical section, $\text{Lock}[\text{pred}_p] = \text{Lock}[q, g] \neq \text{UNLOCKED}$.

Therefore, $\text{Lock}[q, g] \in \mathcal{P} \times \{0, 1\}$ in $C$, so $\text{mylock}_p \in \mathcal{P} \times \{0, 1\}$ when process $p$ performs line 4 or 10 immediately after $C$. Hence, at its next step, process $p$ performs line 8. Let $C'$ be the configuration immediately before $p$ performs line 8. In $C'$, $\text{pred}_p = (q, g)$, so part (2) of Lemma 5.10 implies that $H$ is live and not dormant in $C'$. Since $H$ remains in the remainder section after $C$, $\text{Lock}[q, g] \neq \text{LOCKED}$ in $C'$. Note that Observation 5.2 implies that $\text{Lock}[q, g]$ is not changed between $C$ and $C'$. Thus, when $p$ performs line 8 immediately after $C'$, the CAS succeeds. As a result, $\text{Lock}[q, g]$ is set to LOCKED, which makes $H$ dormant, and $p$ updates $\text{pred}_p$ from $(q, g)$ to $(r, h)$, where $(r, h) = \text{Lock}[q, g]$ in $C'$.

Let $G$ be the current attempt by $(r, h)$ in $C'$. By part (2) of Lemma 5.10, $G < H < I$. Since $p$ performs line 4 or 10 infinitely often, $p$ performs lines 8 and 9 infinitely often.
However, there are only finite number of attempts $J$ such that $J \prec I$. Each time $p$ updates $pred_p$, at least one more of these attempts has become dormant. Therefore, $p$ can update $pred_p$ only finite number of times. This is a contradiction.

\[ \square \]

**5.3 Complexity of this algorithm**

Our algorithm uses $\Theta(N)$ shared variables, each of which has size $\Theta(\log N)$. More precisely, there are $2N + 1$ shared variables, $Tail$ and $Lock[p, f]$, where $p \in P$ and $f$ is Boolean. The size of each shared variable is $2 + \log_2 N$ bits.

In this algorithm, a waiting process $p$ only performs an RMA when the value of $mylock$ has changed or when it performs the cas in line 8. If $q$ is the predecessor of $p$ and $q$ does not abort, then $p$ does not perform line 8. Thus, process $p$ performs $O(1)$ RMAs to enter the critical section if there are no aborts.

If $q$ aborts and re-invokes the trying protocol without its successor $p$ noticing, then when $p$ performs a cas on $Lock[q, f]$, it fails. In [DH04], a failed cas is treated as a read, and they assume that it does not generate an RMA. In [AK01], they consider a CC model in which a failed cas does not generate an RMA as well as other CC models in which it does. In this section, we assume that a failed cas does not generate an RMA. The cas by $p$ succeeds only when its predecessor $q$ aborts and $p$ performs the cas before $q$ re-invokes the trying protocol. When $q$ re-invokes the trying protocol, it is enqueued after $p$. Since $p$ can have at most $N - 1$ different predecessors, there are at most $N - 1$ aborts that can be noticed by $p$. Thus, $p$ performs $O(N)$ successful cas operations before entering the critical section. Therefore, even if there are aborts, each process performs $O(N)$ RMAs to enter the critical section in the CC model using a cache-update protocol.

However, in the CC model using a cache-invalidate protocol, there may be a situation in which a process performs an unbounded number of RMAs. This happens when a
process aborts and re-invokes the trying protocol an unbounded number of times without other processes noticing. For example, suppose that process $q$ updates $\text{Lock}[q, f]$ to $\text{pred}_q$ in line 14 and changes it back to LOCKED in line 1. Although the value of $\text{Lock}[q, f]$ is the same as before $q$ aborted, $\text{Lock}[q, f]$ has been updated. In the CC model with a cache-invalidate protocol, all cached copies of $\text{Lock}[q, f]$ become invalidated. Suppose that another process $p$ busy-waits on $\text{Lock}[q, f]$ but did not notice that $q$ aborted and re-invoked. Then, if $p$ next reads $\text{Lock}[q, f] = \text{LOCKED}$ after $q$ re-invoked, this read is a remote memory access. If $q$ aborts and re-invokes an unbounded number of times and $p$ does not notice any of $q$’s aborts, then $p$ may perform an unbounded number of RMAs in its busy-waiting period.

However, even in the CC model with a cache-invalidate protocol, each abort generates at most one RMA for the waiting process. If a process does not abort, then its successor enters the critical section by performing $O(1)$ RMAs. Hence, the total number of RMAs in an execution is bounded above by a constant times the total number of passages in that execution.

**Theorem 3.** There exists a local-spin abortable mutual exclusion algorithm that uses READ, WRITE, FETCH AND STORE and COMPARE AND SWAP operations and $\Theta(N)$ shared variables, satisfies mutual exclusion, lockout freedom, bounded exit and bounded abort, and has $O(N)$ RMA complexity in the worst case and $O(1)$ RMA complexity in the absence of abort in the CC model with the cache-update protocol.
Chapter 6

S-HAD and Local-spin Abortable Mutual Exclusion

In this chapter, we introduce a new abstract data type, S-HAD (a Sequence with Head, Append, and Delete), from which it is easy to build an abortable mutual exclusion algorithm. We first give the specifications of S-HAD. Next, we present an abortable mutual exclusion algorithm using a single S-HAD object, and prove the correctness of the algorithm. Finally, we discuss the complexity of the algorithm as a function of the complexity of the implementation of S-HAD. We will give two implementations of S-HAD in the following two chapters.

6.1 Definition of S-HAD

An S-HAD is a sequence of elements, each owned by a different process. A process can perform the following operations on an element $R$ that it owns:

- **Head($R$)**: returns TRUE if element $R$ is at the beginning of the sequence.
- **Append($R$)**: appends element $R$ to the end of the sequence.
- **Delete($R$)**: deletes element $R$ from the sequence.

At any point in time, each element occurs at most once in S-HAD, i.e., Append($R$) may
Figure 7 Abortable Mutual Exclusion Algorithm using S-HAD

<table>
<thead>
<tr>
<th>TryingProtocol()</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: ( R := \text{GetNewElement()} )</td>
</tr>
<tr>
<td>T2: Append(( R ))</td>
</tr>
<tr>
<td>T3: while ( \neg \text{Head}(R) ) do</td>
</tr>
<tr>
<td>T4: If the process wants to abort, perform AbortProtocol()</td>
</tr>
<tr>
<td>end while</td>
</tr>
</tbody>
</table>

| ExitProtocol() / AbortProtocol()       |
| E1: \( \text{Delete}(R) \)            |

be called only when \( R \) is not in the sequence, and \( \text{Delete}(R) \) may be called only when \( R \) is in the sequence. Thus, element \( R \) occurs in the sequence if and only if \( \text{Delete}(R) \) has not been performed since \( \text{Append}(R) \) was last performed. \( \text{Head}(R) \) is \( \text{true} \) if and only if \( R \) occurs in the sequence and each element \( X \) that was appended before \( R \) has been deleted from the sequence.

6.2 An abortable mutual exclusion algorithm using an S-HAD object

We can easily build an abortable mutual exclusion algorithm using a linearizable implementation of an S-HAD object. When a process tries to enter the critical section, it appends a new element to the S-HAD object. Then the process keeps performing Head until its element is at the head of the S-HAD object. When Head returns \( \text{true} \), the process enters the critical section. When the process finishes the critical section or wants to abort, it deletes the appended element from the S-HAD object. The detailed algorithms TryingProtocol, ExitProtocol and AbortProtocol appear in Figure 7. GetNewElement is a function that returns a new element. This may be a system call that allocates a memory location for an element or a function that returns an element from a free list.
6.3 Proof of correctness

To prove the correctness of this abortable mutual exclusion algorithm, we show that only the process whose element is at the head of the sequence enters the critical section. We also show that any appended element eventually becomes the head of the sequence if it is not deleted. If process $p$ gets an element $R$ in line T1, we say $\text{owner}(R) = p$.

**Observation 6.1.** If a process $p$ is in the critical section, then the element $R$ at the head of the sequence is owned by $p$.

An operation is *wait-free* if a process performs the operation within a bounded number of its own steps. Since any abortable mutual exclusion algorithm must satisfy the bounded exit and bounded abort properties, Delete must be wait-free. Also, we want the while loop of line T3 to be the only waiting period in the trying protocol, so GetNewElement, Append and Head must be wait-free, too. Given a wait-free GetNewElement and a wait-free implementation of S-HAD, the algorithm in Figure 7 is an FCFS abortable mutual exclusion algorithm.

**Theorem 4.** Given a wait-free implementation of an S-HAD object and a wait-free GetNewElement, the algorithm in Figure 7 is an FCFS abortable mutual exclusion algorithm.

**Proof.** The mutual exclusion property follows from Observation 6.1. Since Delete is wait-free, the algorithm satisfies the bounded abort and bounded exit properties. To prove lockout freedom, suppose that there exists an infinite execution $E$ in which some set of processes, $\mathcal{P}$, keep performing TryingProtocol without entering the critical section or performing AbortProtocol.

Since GetNewElement and Append are wait-free, each process $p$ in $\mathcal{P}$ eventually gets a new element, $R_p$, in line T1 of its last invocation of TryingProtocol, and finishes performing Append($R_p$) on line T2. Since $p$ does not perform ExitProtocol or AbortProtocol after its last invocation of TryingProtocol, $p$ does not subsequently perform Delete($R_p$).
Let $\mathcal{R} = \{ R_p | p \in \mathcal{P} \}$. Let $X$ be the element appended earliest in $\mathcal{R}$, and let $p \in \mathcal{P}$ be the process that performed $\text{Append}(X)$.

By definition, any invocation that last appended an element $S$ before $X$ either eventually enters the critical section and performs $\text{ExitProtocol}$, or eventually performs $\text{AbortProtocol}$. Hence, the invocation eventually performs $\text{Delete}(S)$. Thus, $X$ eventually becomes the head of the sequence, so $\text{Head}(X)$ eventually returns $\text{true}$. Since $p$ keeps performing $\text{TryingProtocol}$ without performing $\text{AbortProtocol}$, it performs $\text{Head}(X)$ infinitely many times. Thus, $p$ will eventually enter the critical section. This contradicts the assumption that $p \in \mathcal{P}$, so the algorithm satisfies lockout freedom.

Since $\text{GetNewElement}$ and $\text{Append}$ are wait-free, each process performs lines T1 and T2 within a bounded number of its own steps. Let the doorway be lines T1 and T2. Then, if process $p$ finishes $\text{Append}(R)$ before process $q$ starts an invocation of $\text{GetNewElement}$ that returns $R'$, then $R$ is appended before $R'$. Thus, if $p$ does not abort, $R$ become the head of the sequence before $R'$. Then, by Observation 6.1, $p$ enters the critical section before $q$. Hence, the resulting abortable mutual exclusion algorithm satisfies the FCFS property.

6.4 Complexity analysis

The complexity of the abortable mutual exclusion algorithm depends on the implementation of S-HAD. First, the resulting abortable mutual exclusion algorithm has bounded exit and abort sections only if the $\text{Delete}$ function is wait-free.

In some systems, allocating a memory location may not be wait-free. However, the algorithm in Figure 7 still solves abortable mutual exclusion if $\text{GetNewElement}$ satisfies the following properties: (1) a process that invokes $\text{GetNewElement}$ eventually completes $\text{GetNewElement}$, and (2) a process that invokes $\text{GetNewElement}$ may return to the remainder section within a bounded number of its own steps if it has to wait during
GetNewElement. (1) is required for lockout freedom and (2) is required for bounded abort.

In order for this algorithm to be local-spin, Head must be carefully implemented. The RMA complexity of one passage is the sum of the RMAs performed during one execution of each of GetNewElement, Append and Delete, and an unbounded number of executions of Head. Thus, in the DSM model, if Head contains even a single RMA, then the resulting algorithm is not local-spin. However, in the CC model, when a process reads a shared variable, it copies its value to its local cache. Hence, even if Head contains remote memory reads, subsequent calls of Head by process \( p \) do not generate RMAs unless another process performs a non-trivial operation on the shared variables \( p \) reads in Head.

In the next two chapters, we present wait-free, linearizable implementations of an S-HAD object shared by \( N \) processes such that any number of calls of Head\((R)\) (by owner\((R)\)) between a call of Append\((R)\) and the subsequent call of Delete\((R)\) generates only a bounded number of remote memory accesses in the CC model. Moreover, if each element is deleted only when it is at the head of the sequence, this number of remote memory accesses is bounded above by a small constant. Our first implementation is simpler but uses unbounded space, and our second implementation uses bounded space.
Chapter 7

A Simple Implementation of S-HAD

In this chapter, we present a simple implementation of S-HAD and prove its correctness. Our implementation is wait-free, but each time a process wants to append an element to the sequence, it uses a new element. Thus, it uses unbounded space. For any element $R$, a process performs $O(N^2)$ RMAs from when it begins $\text{Append}(R)$ until it completes $\text{Delete}(R)$. If all its predecessors are deleted only when they are at the head of the sequence, then the process performs only $O(1)$ RMAs. This implementation gives the first local-spin abortable mutual exclusion algorithm that performs a bounded number of RMAs per passage and performs $O(1)$ RMAs per passage when no aborts occur.

7.1 Algorithm

In this section, we present a simple implementation of an S-HAD object. Detailed pseudo-code is given in Figure 8. Note that the lines are not consecutively numbered. This is so each line has the same number as in the bounded space implementation in Chapter 8.

We begin by explaining the overall structure of the implementation. An S-HAD object is represented by an intree of records, one per element, each with a pointer, $\text{pred}$, which is either NIL or points to another record, and a flag, $\text{del}$. The root of the tree is a dummy record, which is never deleted, whose $\text{del}$ field is always $\text{HEAD}$ and whose $\text{pred}$ field is
always NIL. For every other record $R$, the field $R.del$ indicates whether the element it represents is in the S-HAD sequence or has been logically deleted. The initial value of $R.del$ is false, and it becomes true when the owner of $R$ performs line D1 of Delete($R$). The field $R.pred$ points to another record that was appended before $R$. Thus, the records form an acyclic graph rooted at the dummy record. There is a fetch_and_store (or swap) object, Tail, that initially points to the root. To perform Append($R$), a process atomically reads Tail and updates Tail to point to $R$ in line A2 of Append($R$). Hence, Tail always points to the record that was appended most recently.

When a process wants to know whether the element represented by its record $R$ is at the head of the sequence, it repeatedly updates $R.pred$ until it points to a record that has not been deleted. This is done by Update($R$). Then, the element represented by $R$ is at the head of the sequence if and only if $R.pred$ points to the dummy record.

A process logically deletes its record $R$ by setting $R.del$ to true. Then it calls Update($R$) one more time to ensure that $R$ does not point to another logically deleted record. This is necessary because, otherwise, a sequence with two records that are preceded by arbitrarily many logically deleted records between them and the dummy record can be created by repeatedly deleting the second last record and then appending a new record.

### 7.2 Proof of correctness

At any point during an execution, the state of the S-HAD object is the sequence of records $R$ for which line A2 of Append($R$) has been performed and $R.del = false$. This sequence is ordered by the time at which line A2 of Append($R$) was performed. All records that represent elements in the S-HAD sequence are on the same path to the root and the one that is closest to the root is at the head of the sequence.

We define the linearization point of Append($R$) to be when line A2 is performed.
Figure 8 An Implementation of S-HAD

shared variables:
  type Record ( pred: pointer to a record ∪ { NIL }, initially NIL
    del: { TRUE, FALSE, HEAD }, initially FALSE)
  Record Dummy = (NIL, HEAD)
  Tail: pointer to a record, initially points to Dummy

private variables:
  mypred, ppred: pointer to a record

Head(R:Record) % Precondition: R.del = FALSE, R.pred ≠ NIL
% Postcondition: returns TRUE, if R is the head of the list; otherwise, returns FALSE
  H1: Update(R)
  H2: mypred := R.pred
  H3: return ((*mypred).del = HEAD)

Append(R:Record) % Precondition: R.del = FALSE, R.pred = NIL
  A2: mypred := FETCH_AND_STORE(Tail, &R)
  A3: R.pred := mypred

Delete(R:Record) % Precondition: R.del = FALSE, R.pred ≠ NIL
  D1: R.del := TRUE
  D2: Update(R)

Update(R:Record) % Precondition: R.pred ≠ NIL
  U1: mypred := R.pred
  U2: while (*mypred).del = TRUE do
  U3: ppred := (*mypred).pred
  U5: R.pred := ppred
  U9: mypred := ppred
  end while

Immediately afterwards, R.del = FALSE. Hence, by performing line A2 of Append(R),
the element represented by R is appended to the end of the sequence. The element
represented by R is removed from the sequence when R.del is set to TRUE in line D1. We
define this to be the linearization point of Delete. We define the linearization point of
Head(R) to be when Update(R) returns on line H1, which is when line U2 of Update(R)
is performed with (*mypred).del ≠ TRUE.

Definition 4. Given an execution, we write X ≺ Y to denote that X is the dummy
record or line A2 of Append(X) was performed before line A2 of Append(Y).

Note that ≺ is a total order on the set of records for which line A2 of Append was
performed.

**Definition 5.** \(X\) is the original predecessor of \(Y\) if \(\text{Tail}\) pointed to \(X\) immediately before line A2 of Append\((Y)\) was performed.

Since \(\text{Tail}\) is only accessed by line A2, \(\text{Tail}\) always points to the record whose line A2 was last performed. Thus, if \(X\) is the original predecessor of \(Y\), then \(X \prec Y\). Note that \(\prec\) is the transitive closure of the original predecessor relation.

The following observations come directly from the code.

**Observation 7.1.** If \(R\) is not the dummy record, \(R\.del\) is \texttt{FALSE} until line D1 of Delete\((R)\) is performed. After line D1 of Delete\((R)\), \(R\.del\) remains \texttt{TRUE}.

**Observation 7.2.** Immediately after line A3 of Append\((R)\), \(R\.pred\) points to the original predecessor of \(R\).

**Observation 7.3.** If \(R\) is not the dummy record, \(R\.pred\) is \texttt{NIL} until line A3 of Append\((R)\). After line A3 of Append\((R)\), \(R\.pred\) is never \texttt{NIL}.

**Observation 7.4.** Only the process that owns \(R\) modifies \(R\.pred\). \(R\.pred\) is only changed in line A3 of Append\((R)\) and line U5 of Update\((R)\).

**Observation 7.5.** \(R\.pred = \text{mypred}\) during Update\((R)\) except between lines U5 and U9.

*Proof.* In line U1 of Update\((R)\), \(\text{mypred}\) is set to \(R\.pred\). In line U5 of Update\((R)\), \(R\.pred\) is set to \(ppred\), and then in line U9 of Update\((R)\), \(\text{mypred}\) is also set to \(ppred\). Thus, the values of \(R\.pred\) and \(\text{mypred}\) are the same during Update\((R)\) except between lines U5 and U9. 

To prove correctness, we begin by showing some simple properties and invariants of the implementation.

**Lemma 7.6.** Consider an arbitrary iteration of the while loop in Update\((R)\). Let \(S\) and \(T\) be the records pointed to by \(R\.pred\) immediately before and after line U5 of Update\((R)\)
is performed, respectively. Then, immediately before line U3 of Update(R) was last performed, \(S.\text{pred} \) pointed to \(T\) and \(S.\text{del} = \text{true}\).

**Proof.** Let \(C\) be the configuration immediately before line U3 of Update(\(R\)) was last performed. By Observations 7.4 and 7.5, \(\text{mypred} = \text{R.pred}\) points to \(S\) in \(C\).

Line U5 sets \(\text{R.pred} \) to \(\text{ppred}\), so \(\text{ppred}\) points to \(T\) immediately before line U5. Since \(\text{ppred}\) was last set to (*\(\text{mypred}\).\text{pred}\) in line U3, (*\(\text{mypred}\).\text{pred}\) pointed to \(T\) in \(C\). Since \(\text{mypred}\) points to \(S\) and (*\(\text{mypred}\).\text{pred}\) points to \(T\) in \(C\), \(S.\text{pred}\) points to \(T\) in \(C\).

Also, (*\(\text{mypred}\).\text{del}\) = \text{true} when line U2 was last performed. Thus, \(S.\text{del} = \text{true}\) immediately after line U2. By Observation 7.1, \(S.\text{del} = \text{true}\) in \(C\).

**Invariant 7.7.** In any configuration \(C\) and for any records \(X\) and \(Y\), if \(Y.\text{pred}\) points to \(X\) in \(C\), then \(X \prec Y\).

**Proof.** Let \(C\) be any configuration. Initially, no record points to any other record so the invariant holds when \(C\) is the initial configuration. Assume, for an induction hypothesis, that in any configuration before \(C\), for any records \(X\) and \(Y\), if \(Y.\text{pred}\) points to \(X\), then \(X \prec Y\). Suppose that \(Y.\text{pred}\) points to \(X\) in configuration \(C\). If \(Y.\text{pred}\) points to \(X\) in the previous configuration, then \(X \prec Y\) by the induction hypothesis. Otherwise, by Observation 7.4, either line A3 of Append(\(Y\)) or line U5 of Update(\(Y\)) is performed immediately before \(C\).

In the first case, by Observation 7.2, \(X\) is the immediate predecessor of \(Y\), so \(X \prec Y\) in \(C\).

In the second case, let \(Z\) be the record pointed to by \(Y.\text{pred}\) in the previous configuration. Then, by Lemma 7.6, there exists a configuration before \(C\) in which \(Z.\text{pred}\) points to \(X\). By the induction hypothesis, \(X \prec Z\) and \(Z \prec Y\). Therefore, \(X \prec Y\).

**Corollary 7.8.** In any configuration, \(R.\text{pred}\) does not point to \(R\).
Corollary 7.9. Let $C$ be the configuration immediately before $U5$ of $Update(R)$ is performed, and let $C'$ be the configuration immediately afterwards. Suppose $R.pred$ points to $S$ in $C$ and points to $T$ in $C'$. Then, $S \neq T$.

Proof. Let $B$ be the configuration immediately before line U3 of $Update(R)$ was last performed before $C$. Then by Lemma 7.6, $S.pred$ points to $T$ in $B$. By Corollary 7.8, $S \neq T$.

Corollary 7.10. Let $C$ be the configuration immediately before $U5$ of $Update(R)$ is performed, and let $C'$ be the configuration immediately afterwards. Suppose $R.pred$ points to $S$ in $C$ and points to $T$ in $C'$. Then $T \prec S \prec R$.

Proof. By Lemma 7.6, there exists a configuration in which $S.pred$ points to $T$. By Invariant 7.7, $T \prec S$ and $S \prec R$.

Therefore, for any record, the sequence of records that it points to during the execution decreases with respect to $\prec$.

Invariant 7.11. For any configuration $C$ and any records $X \prec Y \prec Z$, if $Y.del = \text{false}$ in $C$, then $Z.pred$ does not point to $X$ in $C$.

Proof. Suppose, for contradiction, that there exists an execution in which this invariant is violated. Let $C$ be the first configuration of the execution that violates the invariant, and let $C'$ be the previous configuration of $C$. Then, in $C$, there exist records $X \prec Y \prec Z$ such that $Y.del = \text{false}$ and $Z.pred$ points to $X$ in $C$. Note that, by Observation 7.1, $Y.del = \text{false}$ in $C'$, so by the choice of $C$, $Z.pred$ does not point to $X$ in $C'$.

Hence, between $C'$ and $C$, $Z.pred$ is changed to $X$. By Observation 7.4, $Z.pred$ is only changed by line A3 of $Append(Z)$ and line U5 of $Update(Z)$. By Observation 7.2, $Z.pred$ points to its immediate predecessor immediately after line A3 of $Append(Z)$. Since $X$ is not the original predecessor of $Z$, line A3 is not performed between $C'$ and $C$. Thus, line U5 of $Update(Z)$ is performed between $C'$ and $C$. 


Let $W$ be the record pointed to by $Z.pred$ in $C'$. Then, by Lemma 7.6, there exists a configuration $C''$ before $C'$ in which $W.pred$ points to $X$ and $W.del = \text{true}$. By Invariant 7.7, $X \prec W$. By Observation 7.1, $W.del = \text{true}$ in $C'$. Since $Y.del = \text{false}$ in $C'$, $W \neq Y$. If $W \prec Y$, then $W \prec Y \prec Z$ and $Z.pred$ points to $W$ in $C''$, which contradicts the choice of $C$. If $Y \prec W$, then $X \prec Y \prec W$ and $W.pred$ points to $X$ in $C''$, which also contradicts the choice of $C$, since $Y.del = \text{false}$ in $C''$, by Observation 7.1.

Lemma 7.12. Head($R$) returns true if and only if the element represented by $R$ is at the head of the sequence at the linearization point of Head($R$).

Proof. Let $C$ be the configuration at the linearization point of Head($R$), i.e., just after Update($R$) returns on line H1 because (*mypred).del $\neq$ true. Let $S$ be the record pointed to by $R.pred$ in $C$. By Observation 7.5, $R.pred = mypred$ in line U2 of Update($R$). Hence, $S.del \neq$ true in $C$. Also, $S \prec R$ by Invariant 7.7.

By line H2, $R.pred$ and $mypred$ point to the same record in line H3 of Head($R$). Thus, $mypred$ points to $S$ in line H3. If $R$ is at the head of the sequence at the linearization point of Head($R$), then, for all records $X \prec R$ other than the dummy record, $X.del = \text{true}$. Thus, $S$ is the dummy record, so $S.del = \text{head}$. Therefore, Head($R$) returns true.

If $R$ is not at the head of the sequence at the linearization point of Head($R$), there exists a record $T$ which is not the dummy record such that $T \prec R$ and $T.del = \text{false}$. If $S$ was the dummy record, then $S \prec T \prec R$, so by Invariant 7.11, $R.pred$ does not point to $S$ in $C$. This is a contradiction. Hence, $S$ is not the dummy record, so Head($R$) returns false.

Append($R$) is wait-free, since it consists of only two atomic operations. Similarly, Head($R$) and Delete($R$) are wait-free if Update($R$) is wait-free. The following lemma shows that Update is wait-free.

Lemma 7.13. The while loop of Update($R$) is not performed forever.
Proof. Let $S$ be the record pointed to by $R$ immediately before line U5 of Update($R$) is performed and let $T$ be the record pointed to by $R$ immediately after line U5 is performed. By Lemma 7.6, $S.$pred pointed to $T$ immediately before line U3 of Update($R$) was last performed. By Corollary 7.8, $S.$pred never points to $S$, so $S \neq T$. Hence, $R.$pred is updated in each execution of the while loop. By Corollary 7.10, each time $R.$pred is updated, $R.$pred points to an earlier record (with respect to $\prec$). Since the number of records earlier than $R$ is fixed, $R.$pred is not updated infinitely many times. Hence, the while loop is not performed forever. \qed

By the above Lemma, Update($R$) is wait-free and so are Delete($R$) and Head($R$). The correctness of Append and Delete follows from the discussion at the beginning of the section. Lemma 7.12 shows that Head is correct.

**Theorem 5.** The implementation in Figure 8 is a correct, wait-free implementation of S-HAD.

When this implementation of S-HAD is applied to the abortable mutual exclusion algorithm in Figure 7, each process appends a new record for each element. We assume that GetNewElement is a simple system call that allocates a memory location for a new record with $O(1)$ RMAs. Combined with Theorem 4, this shows that the algorithm in Figure 7 using the implementation in Figure 8 is a correct FCFS abortable mutual exclusion algorithm.

### 7.3 Complexity analysis

The space complexity of the implementation is unbounded since each process uses a new record for each Append. However, the number of RMAs performed by a process during each operation is bounded. More precisely, Append takes only two RMAs. The number of RMAs performed during Head and Delete depends on the number of RMAs performed during Update.
We prove that the number of RMAs performed during Update($R$) is bounded above by a function of the number of processes. We say that record $R$ is active if the first line of Append($R$) has been performed, but the last line of Delete($R$) has not yet been performed. Note that, if $R$ is in the sequence, then $R$ is active, but the converse may not hold.

Recall that, by the definition of S-HAD, each process owns at most one active record. Thus it cannot append a new record if there is a record in the sequence that it previously appended, but has not yet deleted.

For any configuration $C$, consider the graph $G(C) = (V(C),E(C))$, where $V(C)$ consists of the dummy record and the set of records $R$ such that line A2 of Append($R$) has been performed, and $E(C)$ consists of the edges $(R,S)$ such that $R$.pred points to $S$ or $mypred$ was assigned to $S$ when line A2 of Append($R$) was performed. We call $G(C)$ as the configuration graph of $C$. Note that each node except the dummy has out-degree one. By Invariant 7.7 and Definition 1, $G(C)$ is acyclic. Thus, $G(C)$ is an in-tree.

**Observation 7.14.** Let $C$ be the configuration immediately after Delete($R$) is completed. In $C$, if $R$.pred points to $S$, then $S$.del $\neq$ TRUE.

**Proof.** The last event of Delete($R$) is line U2 of Update($R$) with (*$mypred$).del $\neq$ TRUE. By Observation 7.5, $mypred = R$.pred in $C$. Thus, in $C$, $S$.del $\neq$ TRUE. \qed

**Invariant 7.15.** In any configuration $C$, if $X$ and $Y$ have the same owner, $Y$ is an ancestor of $X$ in $G(C)$, and both $X$ and $Y$ are inactive, then there exists a record $Z$ on the path from $X$ to $Y$ in $G(C)$ that is active in $C$.

**Proof.** Suppose, for contradiction, that there exists a configuration $C$ in which $X$ and $Y$ have the same owner, $Y$ is an ancestor of $X$ in $G(C)$, both $X$ and $Y$ are inactive, and all records $W$ on the path from $X$ to $Y$ in $G(C)$ are inactive.

Let $X = W_1, \ldots, W_k = Y$ be records on the path from $X$ to $Y$ in $G(C)$. Then, $W_i$.pred points to $W_{i+1}$ in $C$, for $i = 1, \ldots, k - 1$. Since $W_i$ is inactive in $C$, Delete($W_i$)
and, hence, Append($W_i$) were completed before $C$, for all $1 \leq i \leq k$.

By Observation 7.14, immediately after Delete($W_i$) was completed, $W_{i+1}.del \neq \text{TRUE}$. Since the dummy record is not on the path $W_1, \ldots, W_k$, it follows that $W_{i+1}.del = \text{FALSE}$. Thus, $W_{i+1}$ was active when Delete($W_i$) was completed, i.e., Delete($W_i$) was completed before Delete($W_{i+1}$). Therefore, Delete($X$) was completed before Delete($Y$). This is a contradiction, since $X$ and $Y$ have the same owner and $Y \prec X$. 

We say that a record $R$ was deleted prematurely if $R.pre$ did not point to the dummy record when line D1 of Delete($R$) was performed. For the rest of the chapter, we consider an arbitrary record $R$. Let $C$ be the configuration immediately after line A2 of Append($R$) is performed and let $R'$ be the last record that was appended prior to $R$, but was not prematurely deleted. Let $k$ be the number of different processes that appended a record between $R'$ and $R$ inclusive, and let $j$ be the number of active records in $C$ that have been appended prior to $R$ but after $R'$. Let $X_1, \ldots, X_j$ be these active records, where $X_{i+1} \prec X_i$ for $1 \leq i < j$, let $X_0 = R$, and let $X_{j+1} = R'$.

**Lemma 7.16.** For any $0 \leq i \leq j$, there are at most $k - j + i$ records $Z$ such that $X_{i+1} \prec Z \prec X_i$ and $R.pre$ points to $Z$ in some configuration after $C$.

**Proof.** Since Tail always points to a record, immediately after line A3 of Append($R$) is performed, $R.pre$ points to some record $Z_0$. By Observation 7.4 and Corollary 7.9, $R.pre$ is updated when and only when line U5 of Update($R$) is performed. Let $Z_h$ be the record pointed to by $R.pre$ after line U5 of Update($R$) is performed for the $h$th time following $C$. Let $D_h$ be the configuration immediately beforehand, and let $D'_h$ be the configuration immediately afterwards. Then, $R.pre$ points to $Z_{h-1}$ in $D_h$ and $Z_h$ in $D'_h$.

By Corollary 7.10, $Z_h \prec Z_{h-1}$. Let $\hat{D}_h$ be the configuration immediately before line U3 of Update($R$) is last performed prior to $D_h$. In this configuration, $R.pre$ points to $Z_{h-1}$ and $Z_{h-1}.pre$ points to $Z_h$, by Observation 7.4 and Lemma 7.6, respectively.

Let $0 \leq i \leq j$. Suppose there is some record $Z_s$ such that $X_{i+1} \prec Z_s \prec X_i$. If not, we
are done. Let $\ell$ be the smallest number such that $Z_\ell \prec X_i$ and let $m$ be the largest number such that $X_{i+1} \prec Z_m$. Then $X_{i+1} \prec Z_m \prec Z_{m-1} \prec \ldots \prec Z_{\ell+1} \prec Z_\ell \prec X_i \preceq R$. Note that $X_{i+1}$, $X_i$, and $R$ are active in $C$ and all records $Z$ such that $X_{i+1} \prec Z \prec X_i$ are inactive in $C$, by definition of $X_i$. Thus, $Z_\ell, \ldots, Z_m$ are inactive after $C$. Hence, by Observation 7.4, $Z_h, pred$ does not change after $C$. Since $Z_h$ points to $Z_{h+1}$ in $G(\hat{D}_h)$ and $\hat{D}_h$ is after $C$, $Z_h$ also points to $Z_{h+1}$ in $G(C)$. Therefore, in $G(C)$, path $(Z_\ell, \ldots, Z_h, Z_{h+1}, \ldots, Z_m)$ exists. Hence, by Invariant 7.15, each of $Z_\ell, \ldots, Z_m$ has a different owner.

Note that $X_{i+1}, X_{i+2}, \ldots, X_j$ are active in $C$, and $Z_\ell, \ldots, Z_m$ are appended before $C$ and after $X_{i+1}, \ldots, X_j$ are appended. Thus, the owners of $X_{i+1}, \ldots, X_j$ are not the owner of any of $Z_\ell, \ldots, Z_m$. Hence, there are at most $k - j + i$ different owners for $Z_\ell, \ldots, Z_m$. Therefore, $|\{Z_\ell, \ldots, Z_m\}| \leq k - j + i$. 

While $R, pred$ does not change, any sequence of calls to Head($R$) generates at most three RMAs in the CC model: the first time owner($R$) reads $R, pred$ and (*$R, pred, del$), and when (*$R, pred, del$) changes from FALSE to TRUE. If all records are deleted in the same order as they are appended, which is the case for our abortable mutual exclusion when no aborts occur, then $R, pred$ changes only once. Hence, in the abortable mutual exclusion algorithm using this implementation of S-HAD, each process performs $O(1)$ RMAs if no aborts occur.

Now, we prove that the pred pointer of every record changes $O(k^2)$ times.

**Lemma 7.17.** The while loop of Update($R$) is performed at most $k(k+3)/2$ times between beginning Append($R$) and completing Delete($R$).

**Proof.** Note that $j < k$, because $R$ is active in $C$. For any record $A$, if $X_i \prec A \prec R$, then owner($A$) $\neq$ owner($X_i$), since $X_i$ is active in $C$ and $A$ was appended after $X_i$.

By Lemma 7.16, for any $0 \leq i \leq j$, there are at most $k - j + i$ records $Z$ such that $X_{i+1} \prec Z \prec X_i$ and $R, pred$ points to $Z$ in some configuration after $C$. $R, pred$ may point to $X_i$ for $1 \leq i \leq j + 1$ in some configuration after $C$. Also, after $R'$ is deleted from the
head of the sequence, $R_{\text{pred}}$ may point to the dummy record. Hence, $R_{\text{pred}}$ points to at most $(k - j) + (k - j + 1) + \cdots + k + (j + 1) + 1 = (j + 1)(2k - j + 2)/2 + 1$ different records after $C$. Since $j < k$, this quantity is bounded above by $k(k + 3)/2 + 1$.

By Observation 7.4, $R_{\text{pred}}$ is changed only in line A3 of $\text{Append}(R)$ or line U5 of $\text{Update}(R)$. By Corollary 7.10, each time line U5 of $\text{Update}(R)$ is performed, $R_{\text{pred}}$ points to an earlier record. Thus, the number of times line U5 of $\text{Update}(R)$ is performed is one less than the number of different records pointed to by $R_{\text{pred}}$. Therefore, the number of iterations of $\text{Update}(R)$ is at most $k(k + 3)/2$.

The following theorem follows from the previous lemma, since $k \leq N$.

**Theorem 6.** The worst case number of RMAs performed by a process during an invocation of $\text{Append}$ or $\text{Delete}$ in the implementation in Figure 8 is $O(N^2)$.

When a process invokes the abortable mutual exclusion algorithm in Figure 7, it gets a new element $R$ from $\text{GetNewElement}$, and performs $\text{Append}(R)$ and one or more instances of $\text{Head}(R)$ in the trying protocol. When the process performs the abort or exit protocol, it performs $\text{Delete}(R)$. Note that a prematurely deleted record in S-HAD corresponds to an aborting process in the abortable mutual exclusion algorithm.

**Theorem 7.** In the algorithm in Figure 7 using the implementation in Figure 8, each process performs $O(k^2)$ RMAs per passage, where $k$ is the number of processes that began the trying protocol immediately beforehand and subsequently aborted.

**Proof.** Excluding $\text{Update}$, each process performs $O(1)$ RMAs during $\text{Append}$, $\text{Head}$, and $\text{Delete}$. Thus, Lemma 7.17 implies that each process performs $O(k^2)$ RMAs between beginning $\text{Append}(R)$ and completing $\text{Delete}(R)$. Since $\text{GetNewElement}$ takes $O(1)$ RMAs, the algorithm in Figure 7, using the implementation in Figure 8, has $O(k^2)$ RMA complexity.

Since $k$ is bounded by $N$, the worst case RMA complexity of our abortable mutual exclusion algorithm is $O(N^2)$. We show this bound is tight.
Lemma 7.18. There exists an execution of the algorithm in Figure 7 using the implementation in Figure 8 in which \( k' \leq N \) processes abort and some process performs \( k'(k'+1)/2 \) RMAs.

Proof. We construct such an execution in \( k' \) stages. Assume that, after stage \( j-1 \), there are \( N-j+1 \) processes in the remainder section, for \( j = 1, \ldots, k' \). This is true for \( j = 1 \), since, initially, there are \( N \) processes in the remainder section.

Let \( R^j_1, \ldots, R^j_{k'-j+1} \) be \( k' - j + 1 \) different records. Suppose that, at the beginning of stage \( j \), they are appended in this order, each by the trying protocol of a different one of \( k' - j + 1 \) processes. Then \( \text{owner}(R^j_{k'-j}) \), \( \ldots \), \( \text{owner}(R^j_2) \) abort, so that, for \( 2 < i < k' - j + 1 \), \( \text{owner}(R^j_i) \) finishes the abort protocol before \( \text{owner}(R^j_{i-1}) \) starts the abort protocol. Furthermore, \( \text{owner}(R^j_1) \) starts performing the exit or abort protocol after \( \text{owner}(R^j_2) \), \( \ldots \), \( \text{owner}(R^j_{k'-j}) \) all finish the abort protocol. Thus, \( \text{Delete}(R^j_1) \) is finished before \( \text{Delete}(R^j_{i-1}) \) begins, and all of \( \text{Delete}(R^j_1) \), \( \ldots \), \( \text{Delete}(R^j_{k'-j}) \) are finished in stage \( j \). Then, \( R^j_i \) still points to \( R^j_{i-1} \) and \( \text{owner}(R^j_1) \), \( \ldots \), \( \text{owner}(R^j_{k'-j}) \) are all in the remainder section at the end of stage \( j \). At the end of stage \( j \), \( \text{owner}(R^j_{k'-j+1}) \) decides to abort, but does not start performing \( \text{Delete}(R^j_{k'-j+1}) \). Hence, \( \text{owner}(R^j_{k'-j+1}) \) is not in the remainder section at the end of stage \( j \). Therefore, after stage \( j \), \( N - j \) processes are in the remainder section.

Then, after stage \( k' \), \( N - k' \) processes are in the remainder section. Now, suppose that \( \text{owner}(R^k_1) \), \( \ldots \), \( \text{owner}(R^k_{k'-j+1}) \), \( \ldots \), \( \text{owner}(R^k_{k'}) \) begin the abort protocol and perform only the first step of \( \text{Delete}(R^k_1) \), \( \ldots \), \( \text{Delete}(R^k_{k'-j+1}) \), \( \ldots \), \( \text{Delete}(R^k_{k'}) \), respectively. Then, \( k'(k'+1)/2 \) records are logically deleted, but each of them still points to its predecessor. Hence, the path from \( R^k_1 \) to the dummy record has length \( k'(k'+1)/2 + 1 \). Therefore, if \( \text{owner}(R^k_{k'}) \) performs \( \text{Update}(R^k_{k'}) \) at this point, it performs \( k'(k'+1)/2 \) RMAs.

Since \( k' \leq N \), the above lemma implies that our algorithm has \( \Theta(N^2) \) RMA complexity. \(\square\)
**Theorem 8.** There exists a queue-based local-spin abortable mutual exclusion algorithm that uses read, write and fetch-and-store operations, satisfies mutual exclusion, lockout freedom, bounded exit and bounded abort, and has $\Theta(k^2)$ RMA complexity in the CC model, where $k$ is the number of processes that immediately began the trying protocol immediately beforehand and subsequently aborted.
Chapter 8

A Bounded Space Implementation of S-HAD

In this chapter, we present a wait-free implementation of S-HAD that uses bounded space and prove it correct. As part of this, we develop a new reference counting method. The resulting implementation uses $O(N^2)$ space. Like the algorithm in Chapter 7, each process performs $O(N^2)$ RMAs from when it begins an instance of Append($R$) until it next completes an instance of Delete($R$), and it performs $O(1)$ RMAs if all the predecessors of $R$ are deleted only when they are at the head of the sequence. This implementation gives the first local-spin abortable mutual exclusion algorithm that performs a bounded number of RMAs per passage, performs $O(1)$ RMAs per passage when no aborts occur, and uses a bounded amount of space.

In section 8.1, we first explain why existing memory reclamation methods are insufficient for this application. We describe a new memory reclamation method and our algorithm in Section 8.2. The correctness proof and the complexity analysis of our algorithm are presented in Sections 8.3 and 8.4, respectively.
8.1 Memory reclamation

When we apply a memory reclamation method to the algorithm in Chapter 7 to achieve bounded space, we do not want to lose good properties of the algorithm in Chapter 7. In particular, we want a bounded space algorithm that uses standard operations and has the same RMA complexity as the algorithm in Chapter 7.

We need a memory reclamation method that has the following properties. First, memory reclamation should not be done by a dedicated process. Garbage should be reclaimed by existing processes. Second, memory reclamation should be wait-free, i.e., processes do not wait for other processes during memory reclamation. Third, memory reclamation should not increase the RMA complexity of any Head, Append or Delete operation performed by a process in any execution by more than a constant factor. Next, memory reclamation should use only standard atomic operations. In particular, we want to avoid any atomic operations that access more than one shared variable at the same time, since such operations are hard to implement and very few systems support them. Finally, memory reclamation should use $O(\log N)$ bit words.

None of these existing methods satisfy all these requirements. Garbage collection (Section 3.5.2) requires a special process. Also, garbage collection is not wait-free, and the number of RMAs performed by a garbage collector is unbounded. If Hazard pointers [Mic04] (Section 3.5.3) are used with the algorithm in Chapter 7, each process uses $\Theta(N)$ hazard pointers. This is because a process can have $N-1$ deleted nodes that can be in use by different processes. Hence, it takes $\Theta(N)$ RMAs to reclaim one node, which results in more RMAs than the algorithm in the next section. To achieve $O(1)$ amortized RMA complexity for node reclamation, each process must keep a pool of $\Theta(N^2)$ nodes, which results in $\Theta(N^3)$ space complexity. Detlefs et al.’s reference counting method [DMMJ01] (Section 3.5.5) uses DCAS operations, which access two shared variables at the same time. Moreover, neither Detlefs et al.’s nor Valois methods are wait-free. Lernen and Maurer’s reference counting method [LM86] (Section 3.5.7) is for a message passing model and is
not wait-free.

Bevan's reference counting method (Section 3.5.7) can increase the RMA complexity by an arbitrarily large amount. Consider a configuration in which there are three records $X$, $Y$, and $Z$, owned by different processes, such that $Z.pred$ points to $Y$, $Y.pred$ points to $X$, $Y.del = true$, $X.del = Z.pred = false$, and the weight of $Y.pred$ is one. When the owner of $Z$ does pointer jumping, it is supposed to change $Z.pred$ from $Y$ to $X$. However, since the weight of $Y.pred$ is one, $Z.pred$ cannot be set to point to $X$ by copying $Y.pred$. Instead, a new node $W$ is created between $Y$ and $X$, and both $Y.pred$ and $Z.pred$ are set to point to $W$. Since $Z.pred$ cannot point to $X$ directly, the length of a path from $Z$ to the dummy record is not decremented by pointer jumping. Thus, we cannot use Bevan's method with our algorithm.

In Goldberg's reference counting method [Gol89] (Section 3.5.7), each node has an array of size $k$, where the $k$ is the maximum number of generations. However, in the algorithm in Chapter 7, the maximum number of generations can be unbounded. To see why, consider a configuration $C_i$ in which there are three records $X$, $Y$, and $Z$ owned by different processes such that $Z.pred$ points to $Y$, $Y.pred$ points to $X$, $Y.del = true$, $X.del = Z.del = false$, and $Y.pred$ has generation number $i$. Note that $C_0$ can be obtained by having three processes Append records $X$, $Y$, and $Z$ in that order, and then having the owner of $Y$ Delete it. Suppose the owner of $Z$ performs Head, which does pointer jumping, setting $Z.pred$ from $Y$ to $Z$, then the owner of $Y$ Appends a new record $W$, and finally the owner of $Z$ Deletes $Z$ by setting $Z.del = true$. Let $C_{i+1}$ be the resulting configuration. Then $W.pred$ points to $Z$, $Z.pred$ points to $X$, $Z.del = true$, $X.del = W.del = false$, and $Z.pred$ has generation number $i + 1$. Configuration $C_{i+1}$ is the same as $C_i$ except that $Z$ replaces $Y$, $W$ replaces $Z$, and the generation number of the middle pointer is one larger. Hence, by induction, it is possible to reach a configuration in which a pointer has arbitrarily large generation number. Since Goldberg's method requires the generation number to be bounded above by the size of the array at each
node, it cannot be applied to our algorithm.

The specialized memory reclamation methods for Herlihy’s universal constructions [Her91, Her93] (Section 3.5.8) can be used only when each high level operation by a process accesses at most $N$ consecutive nodes of a sequence. Herlihy et al.’s reference counting method [HLM03] (Section 3.5.8) can be used only when all consecutive nodes after some point in a linked list become garbage.

### 8.2 Algorithm

In the algorithm in Chapter 7, even though records are logically deleted from the S-HAD object, processes can still access them to find out that they have been deleted. Also, each time a process performs Append, it uses a new record. Because logically deleted records are not deallocated, that algorithm uses unbounded space. However, after some point in an execution, a logically deleted record is no longer accessed, and we can safely reclaim the memory used by the record. To determine when a logically deleted record is no longer accessed, we use a generalization of reference counts. If the generalized reference count for a record becomes zero, then the record can be physically deleted, and no process is subsequently allowed to access the information in the record.

In our new memory reclamation method, each record has a pointer, $\text{predptr}$, and an original reference counter ($\text{orc}$), which stores an upper bound on the number of pointers in shared memory that point to it. In addition to $\text{orc}$, each record also has two more counters, a proactive reference counter ($\text{prc}$) and a distributed reference counter ($\text{drc}$).

Both $\text{prc}$ and $\text{drc}$ are used to keep track of pointers that have been read and may be written to shared memory in the future.

$R.\text{prc}$ stores the number of times $R.\text{predptr}$ has been read since $R.\text{predptr}$ was last updated. This value is transferred to $S.\text{drc}$ when $R.\text{predptr}$ is changed from pointing to $S$ to pointing to another record. In general, for any record $S$, the sum of the $\text{prc}$’s of
Figure 9 An Implementation of S-HAD with bounded space

shared variables:

type Record (rc: a pair of integers (orc, drc), where 0 ≤ orc < N and
−N < drc < N, initially (0, 0)
pred: a pair (predptr, prc), where predptr is NIL or a pointer to
a record and 0 ≤ prc < N is an integer, initially (NIL, 0)
del: { TRUE, FALSE, HEAD }, initially FALSE
done: { TRUE, FALSE }, initially FALSE)

Record Dummy = ((0,0), (NIL, 0), head, 0)

Tail: pointer to a record, initially points to Dummy

private variables:

mypred, ppred: pointer to a record
myprc, x, y: integer

Head(R :Record) % Precondition: R.del = false, R.pred ≠ (NIL, −)
% Postcondition: returns TRUE, if R is the head of the list; otherwise, returns FALSE
H1: Update(R)
H2: mypred := R.predptr
H3: return ((*mypred).del = HEAD)

Append(R :Record) % Precondition: R.del = false, R.pred = (NIL, 0)
A1: R.rc := (1, 1)
A2: mypred := FETCH_AND_STORE(Tail, &R)
A3: R.pred := (mypred, 0)

Delete(R :Record) % Precondition: R.del = false, R.pred ≠ (NIL, −)
D1: R.del := true
D2: Update(R)
D3: Remove(R)

Update(R :Record) % Precondition: R.pred ≠ (NIL, −)
U1: mypred := R.predptr
U2: while (*mypred).del = true do
U3: (ppred, −) := FETCH_AND_ADD(*mypred).pred, (0, 1))
U4: FETCH_AND_ADD(*ppred).rc, (1, 0))
U5: (−, myprc) := FETCH_AND_STORE(R.pred, (ppred, 0))
U6: (x, y) := FETCH_AND_ADD(*mypred).rc, (−1, myprc − 1))
U7: if (x, y) = (1, 1 − myprc) then % Note that (*mypred).rc = (0, 0)
U8: Remove(*mypred)
end if
U9: mypred := ppred
end while

Remove(R :Record)
R1: if TEST_AND_SET(R.done) = true then
R2: (mypred, myprc) := FETCH_AND_STORE(R.pred, (NIL, 0))
R3: (x, y) := FETCH_AND_ADD(*mypred).rc, (−1, myprc − 1))
R4: recycle(R)
R5: if (x, y) = (1, 1 − myprc) then % Note that (*mypred).rc = (0, 0)
R6: Remove(*mypred)
end if
end if
all records that point to $S$ plus $S.drc$ is at most the number of times a pointer to $S$ has been read minus the number of times a pointer to $S$ has been overwritten.

$R.prc$ is stored together with $R.predptr$ in a single variable $R.pred$, so that they can be accessed together. We first show that $prc \leq N$, so $prc$ can be represented using $\lceil \log_2 N \rceil$ bits. When a process does pointer jumping by reading a pointer $X.pred$ during Head($Z$), then all records that appended later than $Z$ will not access $X.pred$ before $X.pred$ is changed. If a process reads $X.pred$ during Delete($Y$), finishes Delete($Y$), Appends a new record $Y'$, and reads $X.pred$ again, then this read happens during Head($Y'$), since all records appended between $X$ and $Y'$ are set to deleted. Thus, while $X.pred$ is not changed, at most $N - 1$ processes can read $X.pred$ during Delete. Therefore, in total, $N$ processes can read $X.pred$ while $X.pred$ is not updated, so $X.prc \leq N$.

We will also show in Lemma 8.22 that less than $4N^2$ records are used, so a pointer can be represented using $2\lceil \log_2 N \rceil + 2$ bits if the records are all allocated from one block of memory. Hence, $pred = (predptr, prc)$ can be represented using $3\lceil \log_2 N \rceil + 2$ bits, where the lower $\lceil \log_2 N \rceil$ bits of $pred$ are used for $prc$. Provided $prc < N - k$, FETCH\_AND\_ADD$(pred, k)$ adds $k$ to $prc$ without changing $predptr$. In our algorithm, whenever $predptr$ is set to point to a record $X$, $prc$ becomes zero, which can be accomplished by performing a FETCH\_AND\_STORE operation on $pred$ with the address of $X$ followed by a string of $\lceil \log_2 N \rceil$ zeros. Storing a pointer together with a counter was also done in [Gol89] and [HLM03].

Similarly $R.drc$ is stored together with $R.orc$ in a single variable $R.rc$. We will prove that the range of $orc$ is from 0 to $N - 1$ and the range of $drc$ is from $1 - N$ to $N - 1$. Hence, $rc = (orc, drc)$ can be represented using $2\lceil \log_2 N \rceil + 1$ bits in a single word of memory. FETCH\_AND\_ADD$(rc, (m, n))$ can be simulated by FETCH\_AND\_ADD$(rc, m \cdot 2^{\lceil \log_2 N \rceil + 1} + n)$.

The algorithm does not need to know the value of $N$, provided the word size $B$ is sufficiently large (and known). The highest order two thirds of the bits of $pred$ can be used for $predptr$ and the other one third of the bits can be used for $prc$. When a process sets
predptr to point to a record $X$ and prc to 0, it performs a \texttt{fetch\_and\_store} operation on pred with the address of $X$ followed by a string of $\left\lfloor \frac{B}{3} \right\rfloor$ zeros. One half of the bits of rc can be used for orc, the other half can be used for drc, and \texttt{fetch\_and\_add}(rc, (m, n)) is simulated by \texttt{fetch\_and\_add}(rc, m \cdot 2^{\left\lfloor \frac{B}{2} \right\rfloor} + n).

Pseudo-code for the algorithm is presented in Figure 9. The newly added lines (lines A1, D3, U4, U6, U7, U8) are shaded. Note that these lines only deal with rc (in lines A1, U4, U6 and U7) or call Remove (in lines D3 and U8). Lines H1, H3, A2, D1, D2, U2, and U9 are exactly the same as in Chapter 7. \texttt{R.predptr} in Figure 9 stores \texttt{R}'s pointer, which is the same as \texttt{R.pred} in Chapter 7. Thus, lines H2 and U1 are the same as the corresponding lines in Chapter 7.

\texttt{Head}(R) is the same as in Chapter 7. In Append(R), \texttt{owner}(R) sets \texttt{R.rc} to (1,1) before appending \texttt{R} to the end of the sequence. In line A3 of Append(R), \texttt{R.predptr} is set to \texttt{mypred}, which is the same as in Chapter 7, and \texttt{R.prc} is set to 0. Thus, except for dealing with rc and prc, Append(R) in Figure 9 is the same as in Chapter 7. Delete(R) is the same as the algorithm in Chapter 7 except that Remove(R) is called at the end.

Most of the differences are inside the while loop of Update(R). In line U3, \texttt{ppred} is set to (*\texttt{mypred}).\texttt{predptr}, which is the same as in Chapter 7, and (*\texttt{mypred}).\texttt{prc} is incremented. Note that (\texttt{ppred}, -) means that the second field of the returned value (i.e., prc) is ignored. In line U5, \texttt{R.predptr} is set to \texttt{ppred}, which is the same as in Chapter 7, \texttt{myprc} is set to \texttt{R.prc}, and \texttt{R.prc} is set to 0. Also, note that (-, \texttt{myprc}) means the first field of the returned value is ignored. Thus, except for dealing with rc and prc and calling Remove, Update(R) is the same as in Chapter 7.

In the algorithm in Chapter 7, \texttt{R.pred} can only be changed by \texttt{owner}(R). However, in the algorithm in Figure 9, \texttt{R.predptr} can also be changed by other processes in line R2, but only after \texttt{R} has been logically deleted. This does not affect the RMA complexity of Head(R), which is only performed while \texttt{R} is in the sequence.

To understand the effect of the modifications to Update(R), let’s consider the situation
when process $p$, which owns record $R$, wants to update $R$'s predecessor pointer to point to the predecessor of its predecessor, i.e., $R.p\text{predptr} := (*R.p\text{predptr}).\text{predptr}$. Suppose $X$ is $R$’s predecessor, $Y$ is $X$’s predecessor, and $p$’s local variable $\text{mypred}$ points to $X$. To change $R$ to point from $X$ to $Y$, process $p$ performs line U3, in which $p$ atomically reads $X.p\text{predptr}$ and increments $X.p\text{rc}$ using FETCH\_AND\_ADD. Incrementing $X.p\text{rc}$ means that $R$ will reference $Y$ and it learned about $Y$ from $X$. Next, $p$ increments $Y.\text{orc}$ on line U4. On line U5, $p$ atomically changes $R.p\text{predptr}$ to $Y$, reads $R.p\text{rc}$ into its local variable $\text{myprc}$, and resets $R.p\text{rc}$ to 0, using FETCH\_AND\_STORE. Hence, $\text{myprc}$ stores the number of processes that have accessed $R.p\text{predptr}$ between the last two updates of $R.p\text{predptr}$. Finally, on line U6, $p$ atomically decrements $X.\text{orc}$ and adds $\text{myprc} - 1$ to $X.d\text{rc}$, using FETCH\_AND\_ADD. Note that, by adding $\text{myprc} - 1$ to $X.d\text{rc}$, the value that had been stored in $R.p\text{rc}$ before it was reset is transferred to $X.d\text{rc}$ and $X.d\text{rc}$ is decremented to reflect that $R$ is no longer pointing to $X$.

To physically delete a record $R$, a process calls recycle($R$), which is either a system call that deallocates $R$’s memory or a procedure that moves $R$ to a free list. A record $R$ can be physically deleted only when no records point to $R$, no records will point to $R$, and Delete($R$) has been completed. $R.\text{orc} = 0$ indicates that no record currently points to $R$, and $R.d\text{rc} = 0$ indicates that no records will point to $R$. Hence, when $R.rc = (0, 0)$ and Delete($R$) is completed, $R$ can be physically deleted. To ensure that these conditions are met, the procedure Remove($R$) is called twice: one by the owner of $R$ at the end of Delete($R$) (line D3) and the other by a process that finds $R.rc = (0, 0)$ during Update (line U8) or Remove (line R6). Remove($R$) is called from exactly one of line U8 or line R6, so Remove($R$) is called exactly twice. The first time Remove($R$) is called, it does nothing. The second time, it physically deletes $R$ by performing recycle($R$) on line R4. To accomplish this, a TEST\_AND\_SET object, $R.d\text{one}$ is used, and the rest of Remove is only performed if it returns TRUE. Note that record $R$ can be physically deleted by any process, although Delete($R$) can be called only by the owner of $R$. 
Remove is called recursively if physically deleting a record causes another record’s reference counts to become \((0, 0)\): When a process physically deletes a record \(R\), it also removes its pointer, \(R.\text{predptr}\). If \(R.\text{predptr}\) pointed to another record \(S\), then \(S\)’s reference counts must be updated. This may cause \(S.\text{rc}\) to become \((0, 0)\) and, if it is, \(\text{Remove}(S)\) is called recursively in line R6. These recursive calls add only \(O(k^2)\) RMAs in total, if \(k\) is the number of processes that appended records before \(R\) and deleted them prematurely. Hence, they do not affect the overall asymptotic RMA complexity of the algorithm.

Unlike the reference counting in [Val95], our algorithm allows each record to be reclaimed by the system, provided the system calls for memory allocation and deallocation each take \(O(1)\) RMAs. In this case, \(\text{GetNewRecord}\) in Figure 7 is a system call that allocates memory and \(\text{recycle}(R)\) on line R4 of Figure 9 is a system call that deallocates memory.

Alternatively, we can use a free list of length at most \(3N\) for each process. In Section 8.3, Lemma 8.21 shows why \(3N\) records suffice for each process. Each process, \(p\), maintains a Boolean array of size \(3N\), which indicates which records are available. To get a new record, process \(p\) keeps checking each element of the array until it finds a true bit. If the \(i\)th bit in the array is \text{true}, \(p\) sets it to \text{false} and uses its \(i\)th record. When some process recycles the \(i\)th record of \(p\), it sets the \(i\)th element of \(p\)’s array to \text{true}. Since \(p\) is the only process that sets elements of its array to \text{false}, a remote memory access is not generated when \(p\) reads \text{false}. Therefore, both \(\text{GetNewRecord}\) in Figure 7 and \(\text{recycle}(R)\) on line R4 of Figure 9 generate only \(O(1)\) RMAs.

Our reference counting method can be used when a node can have multiple pointers instead of just one. In this case, when a process removes a node, it updates the value of \(rc\) for each node that it pointed to using the value of \(prc\) associated with the pointer to that node. As is the case for other reference counting methods, a cycle of deleted nodes cannot be reclaimed.
8.3 Proof of correctness

Our reference counting method only changes records in line R2 of Remove(R), which sets \( R.predptr \) to NIL, and line R4 of Remove(R), which frees (i.e., physically deletes) \( R \).

In this section, we prove that, after these lines are performed, no process will access \( R \). We begin with some definitions and present some simple observations. Then we prove some invariants involving \( orc \), \( drc \) and \( prc \). In particular, Invariant 8.11 describes what \( orc \) stores and Invariant 8.12 describes the relationship between \( prc \) and \( drc \). Then we prove that, after Remove(\( R \)) has been called twice, \( R \) has been freed and no process will access \( R \). Finally we show that, using this reference counting method, our algorithm is a correct implementation of S-HAD.

We say that \( X \) points to \( Y \) or \( X.pred \) points to \( Y \) when the first field of \( X.pred \) is \&\( Y \). If the next event by the owner of \( R \) is line \( L \), then we say that \( owner(\( R \)) \) is at line \( L \). We also say that \( R \) is live if \( R \) has been Appended (i.e., line A3 of Append(\( R \)) has been performed.) but it has not subsequently been freed.

Observation 8.1. Only the process that owns \( R \) modifies \( R.predptr \) in line A3 of Append(\( R \)) and line U5 of Update(\( R \)). Immediately after line A3 of Append(\( R \)) or line U5 of Update(\( R \)), \( R.prc \) is set to 0. Other processes can modify \( R.prc \) only in line U3 and only if \((\ast mypred) = \& R \).  

Consider any configuration \( C \) in which some record \( R \) is in the sequence and \( Tail \) does not point to \( R \). Let \( A_{R}(C) \) be the set of records \( Z \) such that \( Z.pred \) points to \( R \) in \( C \). Let \( B_{R}(C) \) be the set of active records whose owners are at line U4 with \( ppred = \& R \). Let \( C_{R}(C) \) be the set of active records whose owners are at line U5 with \( ppred = \& R \). Let \( D_{R}(C) \) be the set of active records whose owners are at line U6 with \( mypred = \& R \). Finally, let \( E_{R}(C) \) be the set of active records whose owners are at line R3 with \( mypred = \& R \).

Observation 8.2. \( B_{R}(C) \), \( C_{R}(C) \), \( D_{R}(C) \), and \( E_{R}(C) \) are disjoint sets.
Lemma 8.3. $A_R(C)$ is disjoint from $B_R(C)$, $C_R(C)$, and $D_R(C)$.

Proof. If $Z \in B_R(C) \cup C_R(C)$, then when line U3 of Update($Z$) was last performed, $ppred$ was set to $\& R$. Hence, immediately before this line U3 was last performed, $(\ast mypred).pred = \& R$. By Corollary 7.8, $mypred \neq \& R$. By Observation 7.5, $Z.predptr = mypred$ when $Z$ is at lines U3, U4, and U5. Since both $mypred$ and $Z.predptr$ are not changed until U5 is performed, $Z.predptr \neq \& R$ when $Z$ is at lines U3, U4, and U5. Thus, $Z \notin A_R(C)$.

If $Z \in D_R(C)$, then $mypred = \& R$, so immediately before line U3 of Update($Z$) was last performed, $(\ast mypred).predptr = R.predptr \neq \& R$ by Corollary 7.8. Hence, immediately after line U3 of Update($Z$) was last performed, $ppred \neq \& R$. When line U5 of Update($Z$) was last performed, $R.predptr$ was set to $ppred$, so $R.predptr \neq \& R$. Hence, $Z \notin A_R(C)$.

Observation 8.4. $Z$ is at line U5 with $mypred = \& R$ in $C$ if and only if $Z \in A_R(C)$.

Proof. By Observation 7.5, $Z.predptr = mypred$ when $Z$ is at line U5. Hence, $Z$ is at line U5 with $mypred = \& R$ in $C$ if and only if $Z.predptr = \& R$, that is, $Z \in A_R(C)$.

The following observations show how each set is updated by each event. Let $C'$ be the configuration immediately after $C$.

Observation 8.5. If line U4 of Update($Z$) is performed with $ppred = \& R$ between $C$ and $C'$, then $R.orc$ is incremented, $A_R(C') = A_R(C)$, $B_R(C') = B_R(C) - \{Z\}$, $C_R(C') = C_R(C) + \{Z\}$, $D_R(C') = D_R(C)$, and $E_R(C') = E_R(C)$.

Proof. Since $ppred = \& R$, this event adds 1 to $(\ast ppred).orc = R.orc$. In $C$, $Z$ is at line U4 with $ppred = \& R$, so $Z \in B_R(C)$. In $C'$, $Z$ is at line U5 with $ppred = \& R$, so $Z \in C_R(C')$. By Observation 8.2 and Lemma 8.3, $B_R(C') = B_R(C) - \{Z\}$, and $C_R(C') = C_R(C) + \{Z\}$. $A_R$, $D_R$, and $E_R$ are not changed between $C$ and $C'$.
Observation 8.6. If line U5 of Update(Z) is performed with ppred = &R between C and C', then Z.prc is set to 0, $A_R(C') = A_R(C) + \{Z\}$, $B_R(C') = B_R(C)$, $C_R(C') = C_R(C) - \{Z\}$, $D_R(C') = D_R(C)$, and $E_R(C') = E_R(C)$.

Proof. Since ppred = &R, this event sets Z.pred to (&R, 0). In C, Z is at line U5 with ppred = &R, so Z ∈ $A_R(C)$. In C', Z.pred points to R, so Z ∈ $A_R(C')$. By Observation 8.2 and Lemma 8.3, $A_R(C') = A_R(C) + \{Z\}$, and $C_R(C') = C_R(C) - \{Z\}$. $B_R$, $D_R$, and $E_R$ are not changed between C and C'.

Observation 8.7. If $Z \in A_R(C)$ and line U5 of Update(Z) is performed between C and C', then Z.prc is set to 0, $A_R(C') = A_R(C) - \{Z\}$, $B_R(C') = B_R(C)$, $C_R(C') = C_R(C)$, $D_R(C') = D_R(C) + \{Z\}$, and $E_R(C') = E_R(C)$.

Proof. Since $Z \in A_R(C)$, $Z \notin C_R(C)$ by Observation 8.2. Hence, this event sets Z.predptr to some record other than R. Thus, $Z \notin A_R(C')$. By Observation 8.4, mypred = R.predptr = &R in C. Thus, in C', Z is at line U6 with mypred = &R, so $Z \in D_R(C')$. By Observation 8.2 and Lemma 8.3, $A_R(C') = A_R(C) - \{Z\}$, and $D_R(C') = D_R(C) + \{Z\}$. $B_R$, $C_R$, and $E_R$ are not changed between C and C'.

Observation 8.8. If line U6 of Update(Z) is performed with mypred = &R between C and C', then R.orc is decremented, myprc − 1 is added to R.drc, $A_R(C') = A_R(C)$, $B_R(C') = B_R(C)$, $C_R(C') = C_R(C)$, $D_R(C') = D_R(C) - \{Z\}$, and $E_R(C') = E_R(C)$.

Proof. Since mypred = &R, this event adds (−1, myprc − 1) to (*mypred).rc = R.rc. In C, Z is at line U6 with mypred = &R, so $Z \in D_R(C)$. In C', Z is at line U7, so $Z \notin B_R(C') \cup C_R(C') \cup D_R(C') \cup E_R(C')$. By Observation 8.2 and Lemma 8.3, $Z \notin A_R(C)$, and Z.pred stays the same in C', so $Z \notin A_R(C')$. Hence, $D_R(C') = D_R(C) - \{Z\}$, and $A_R$, $B_R$, $C_R$ and $E_R$ are not changed between C and C'.

Observation 8.9. If $Z \in A_R(C)$ and line R2 of Remove(Z) is performed by owner(W) between C and C', then Z.prc is set to 0, $A_R(C') = A_R(C) - \{Z\}$, $B_R(C') = B_R(C)$, $C_R(C') = C_R(C)$, $D_R(C') = D_R(C)$, and $E_R(C') = E_R(C) + \{W\}$. 
Proof. Since $Z \in A_R(C)$, this event sets $Z, \text{pred}$ to (NIL, 0). Thus, $Z \notin A_R(C')$. Also this event sets mypred to &R. Thus, in $C'$, $Z$ is at line R3 with mypred = &R. Since owner(W) performs Remove(Z), $W \in E_R(C')$. By Observation 8.2 and Lemma 8.3, $A_R(C') = A_R(C) - \{Z\}$, and $E_R(C') = E_R(C) + \{W\}$. $B_R$, $C_R$, and $D_R$ are not changed between $C$ and $C'$.

Observation 8.10. If line R3 of Remove(Z) is performed with mypred = &R by owner(W) between $C$ and $C'$, then $R,orc$ is decremented, myprc − 1 is added to $R,drc$, $A_R(C') = A_R(C)$, $B_R(C') = B_R(C)$, $C_R(C') = C_R(C)$, $D_R(C') = D_R(C)$, and $E_R(C') = E_R(C) - \{W\}$.

Proof. Since mypred = &R, this event adds (−1, myprc − 1) to (*mypred).rc = $R.rc$. In $C$, $Z$ is at line R3 with mypred = &R. Since owner(W) performs Remove(Z), $W \in E_R(C)$. In $C'$, $Z$ is at line R4, so $W \notin E_R(C')$. Hence, $E_R(C') = E_R(C) - \{W\}$, and $A_R$, $B_R$, $C_R$ and $D_R$ are not changed between $C$ and $C'$.

$R,orc$ is the number of records that are currently pointing to $R$, plus the number of records that have just incremented $R,orc$ and are about to point to $R$ (i.e., between lines U4 and U5), plus the number of records that are about to decrement $R,orc$ after changing their pointer away from $R$ (i.e., between lines U5 and U6 or between lines R2 and R3). Invariant 8.11 describes this more formally.

Invariant 8.11. In any configuration $C$, if record $R$ is live and Tail does not point to $R$, then $R,orc = |A_R(C)| + |C_R(C)| + |D_R(C)| + |E_R(C)|$.

Proof. When $R$ is appended to the sequence, $R,rc$ is set to (1, 1) in line A1 of Append($R$). When the successor $Z$ of $R$ in the sequence is appended, $Z,\text{pred}$ is set to $R$ in line A3 of Append($Z$). Thus, immediately after $R$ becomes not the end of the sequence, $R,orc = 1$, $A_R(C) = \{Z\}$, and $C_R(C) = D_R(C) = E_R(C) = \emptyset$ Thus, $R,orc = |A_R(C)| + |C_R(C)| + |D_R(C)| + |E_R(C)| = 1.$
Chapter 8. A Bounded Space Implementation of S-HAD

Now suppose that the invariant holds in some configuration \( C \), and let \( C' \) be the configuration immediately after \( C \). Let \( k = R.orc = |A_R(C)| + |C_R(C)| + |D_R(C)| + |E_R(C)| \) in \( C \). First consider the case when \( R.rc \) is changed between \( C \) and \( C' \). \( R.rc \) is changed by line U4 with \( ppred = &R \), U6 with \( mypred = &R \), and R3 with \( mypred = &R \).

Case 1: Suppose there exists some record \( Z \) such that line U4 of Update(\( Z \)) is performed with \( ppred = &R \) between \( C \) and \( C' \). By Observation 8.5, \( R.orc \) is incremented, \( |A_R(C')| = |A_R(C)|, |C_R(C')| = |C_R(C)| + 1, |D_R(C')| = |D_R(C)|, \) and \( |E_R(C')| = |E_R(C)| \). Therefore, in \( C' \), \( R.orc = k + 1 = |A_R(C)| + |C_R(C)| + |D_R(C)| + |E_R(C)| + 1 = |A_R(C')| + |C_R(C')| + |D_R(C')| + |E_R(C')| \).

Case 2: Suppose there exists some record \( Z \) such that line U6 of Update(\( Z \)) is performed with \( mypred = &R \) between \( C \) and \( C' \). By Observation 8.8, \( R.orc \) is decremented, \( |A_R(C')| = |A_R(C)|, |C_R(C')| = |C_R(C)|, |D_R(C')| = |D_R(C)| - 1, \) and \( |E_R(C')| = |E_R(C)| \). Therefore, in \( C' \), \( R.orc = k - 1 = |A_R(C)| + |C_R(C)| + |D_R(C)| + |E_R(C)| - 1 = |A_R(C')| + |C_R(C')| + |D_R(C')| + |E_R(C')| \).

Case 3: Suppose there exists some process \( Z \) such that line R3 of Remove(\( Z \)) is performed with \( mypred = &R \) between \( C \) and \( C' \). By Observation 8.10, \( R.orc \) is decremented, \( |A_R(C')| = |A_R(C)|, |C_R(C')| = |C_R(C)|, |D_R(C')| = |D_R(C)|, \) and \( |E_R(C')| = |E_R(C)| - 1 \). Therefore, in \( C' \), \( R.orc = k - 1 = |A_R(C)| + |C_R(C)| + |D_R(C)| + |E_R(C)| - 1 = |A_R(C')| + |C_R(C')| + |D_R(C')| + |E_R(C')| \).

Case 4: Suppose there exists some process \( Z \) such that \( Z \in A_R(C) \) and \( Z \notin A_R(C') \). If \( Z.pred \) points to \( R \) in \( C \) and does not point to \( R \) in \( C' \), then the first field of \( Z.pred \) is updated between \( C \) and \( C' \). This happens when either line U5 of Update(\( Z \)) or line R2 of Remove(\( Z \)) is performed between \( C \) and \( C' \).

Case 4-1: line U5 of Update(\( Z \)) is performed between \( C \) and \( C' \). By Observation 8.7, \( |A_R(C')| = |A_R(C)| - 1, |C_R(C')| = |C_R(C)|, |D_R(C')| = |D_R(C)| + 1, \) and \( |E_R(C')| = |E_R(C)| \).
\(|E_R(C)|\). \(R.rc\) is not changed between \(C\) and \(C'\). Therefore, in \(C'\), \(R.orc = k = |A_R(C)| + |C_R(C')| + |D_R(C')| + |E_R(C')|\).

Case 4-2: line \(R2\) of \(\text{Remove}(Z)\) is performed between \(C\) and \(C'\). By Observation 8.9, \(|A_R(C')| = |A_R(C)| - 1\), \(|C_R(C')| = |C_R(C')|\), \(|D_R(C')| = |D_R(C')|\), and \(|E_R(C')| = |E_R(C')| + 1\). \(R.rc\) is not changed between \(C\) and \(C'\). Therefore, in \(C'\), \(R.orc = k = |A_R(C)| + |C_R(C)| + |D_R(C)| + |E_R(C)| = |A_R(C')| + |C_R(C')| + |D_R(C')| + |E_R(C')|\).

Case 5: Suppose there exists some process \(Z\) that \(Z \notin A_R(C)\) and \(Z \in A_R(C')\). This happens when line \(U5\) of \(\text{Update}(Z)\) with \(ppred = \&R\) is performed between \(C\) and \(C'\). By Observation 8.6, \(|A_R(C')| = |A_R(C)| + 1\), \(|C_R(C')| = |C_R(C')| - 1\), \(|D_R(C')| = |D_R(C')|\), and \(|E_R(C')| = |E_R(C')|\). \(R.rc\) is not changed between \(C\) and \(C'\). Therefore, in \(C'\), \(R.orc = k = |A_R(C)| + |C_R(C)| + |D_R(C)| + |E_R(C)| = |A_R(C')| + |C_R(C')| + |D_R(C')| + |E_R(C')|\).

Case 6: Suppose there exists some process \(Z\) such that \(Z \in C_R(C)\) but \(Z \notin C_R(C')\). Then, line \(U5\) of \(\text{Update}(Z)\) is performed with \(ppred = \&R\) between \(C\) and \(C'\). This case is the same as Case 5.

Case 7: Suppose there exists some process \(Z\) such that \(Z \notin C_R(C)\) but \(Z \in C_R(C')\). Then, line \(U4\) of \(\text{Update}(Z)\) is performed with \(ppred = \&R\) between \(C\) and \(C'\). This case is the same as Case 1.

Case 8: Suppose there exists some process \(Z\) such that \(Z \in D_R(C)\) but \(Z \notin D_R(C')\). Then, line \(U6\) of \(\text{Update}(Z)\) is performed with \(mypred = \&R\) between \(C\) and \(C'\). This case is the same as Case 2.

Case 9: Suppose there exists some process \(Z\) such that \(Z \notin D_R(C)\) but \(Z \in D_R(C')\). Then, \(Z\) is at line \(U5\) in \(C\). Since \(mypred = \&R\) in \(C\), \(Z \in A_R(C)\) by Observation 8.4. This case is the same as Case 4-1.

Case 10: Suppose there exists some process \(Z\) such that \(Z \in E_R(C)\) but \(Z \notin E_R(C')\). This happens when line \(R3\) of \(\text{Remove}(R)\) is performed with \(mypred \& R\) between \(C\) and
C'. This case is the same as Case 3.

Case 11: Suppose there exists some process Z such that $Z \notin E_R(C)$ but $Z \in E_R(C')$. This happens when line R2 of Remove(R) is performed between C and C'. This case is the same as Case 4-2.

The sum of $R.drc$, the proactive counts of R’s successor, and the values of $\text{myprc}$ of the records that was R’s successor but has not added its proactive counts to $R.drc$ (i.e., they are between lines U5 and U6 or between lines R2 and R3) is the number of records that are currently pointing to R plus the number of active records whose owners have read & R, but are not yet pointing to R (i.e., between lines U3 and U5) plus the number of records between lines U5 and U6 or between lines R2 and R3. Invariant 8.12 describes this more formally.

Invariant 8.12. For any record $Z \in \mathbb{A}_R(C)$, let $x(Z)$ denote the value of $Z.prc$, and for any record $Z \in \mathbb{D}_R(C) \cup \mathbb{E}_R(C)$, let $x(Z)$ denote the value of $\text{myprc}$, i.e. the second component returned on line U5 or R2, respectively. Let $\text{sum}_R(C) = \sum \{ x(Z) | Z \in \mathbb{A}_R(C) \} + \sum \{ x(Z) | Z \in \mathbb{D}_R(C) \cup \mathbb{E}_R(C) \}$. Then, in any configuration C, if record R is live and Tail does not point to R, then $R.drc + \text{sum}_R(C) - |\mathbb{A}_R(C)| - |\mathbb{D}_R(C)| - |\mathbb{E}_R(C)| = |\mathbb{B}_R(C)| + |\mathbb{C}_R(C)|$.

Proof. Suppose that R is live but not at the end of the sequence in C. When R is appended to the sequence, $R.rc$ is set to (1, 1) in line A1 of Append(R). When the successor Z of R in the sequence is appended, $Z.prd$ is set to ($\&$R, 0) in line A3 of Append(Z). Thus, in the resulting configuration, $C$, $\mathbb{A}_R(C) = \{ Z \}$, $\mathbb{B}_R(C) = \mathbb{C}_R(C) = \mathbb{D}_R(C) = \mathbb{E}_R(C) = \emptyset$, and $\text{sum}_R(C) = 0$. Hence, $R.drc + \text{sum}_R(C) - |\mathbb{A}_R(C)| - |\mathbb{D}_R(C)| - |\mathbb{E}_R(C)| = |\mathbb{B}_R(C)| + |\mathbb{C}_R(C)| = 0$.

Now suppose that the invariant holds in some configuration C, and let $C'$ be the configuration immediately after C. Let $k = R.drc$ in C. Then, $k + \text{sum}_R(C) - |\mathbb{A}_R(C)| - |\mathbb{D}_R(C)| - |\mathbb{E}_R(C)| = |\mathbb{B}_R(C)| + |\mathbb{C}_R(C)|$. 

Case 1: First consider the case when $R.drc$ is changed between $C$ and $C'$. $R.drc$ is changed in line U6 with $mypred = &R$ and line R3 with $mypred = &R$. Let $Z$ be a record such that line U6 of Update($Z$) or R3 of Remove($Z$) is performed with $mypred = &R$ and $myprc = v$ between $C$ and $C'$, for some $v \in \mathbb{Z}$.

Case 1-1: Suppose line U6 Update($Z$) is performed with $mypred = &R$ between $C$ and $C'$. By Observation 8.8, $y - 1$ is added to $R.drc$, $|\mathbb{A}_R(C')| = |\mathbb{A}_R(C)|$, $|\mathbb{B}_R(C')| = |\mathbb{B}_R(C)|$, $|\mathbb{C}_R(C')| = |\mathbb{C}_R(C)|$, $|\mathbb{D}_R(C')| = |\mathbb{D}_R(C)| - 1$, and $|\mathbb{E}_R(C')| = |\mathbb{E}_R(C)|$. Since $Z \notin \mathbb{A}_R(C') \cup \mathbb{D}_R(C') \cup \mathbb{E}_R(C')$, $sum_R(C') = sum_R(C) - y$. Hence, in $C'$, $R.drc + sum_R(C') - |\mathbb{A}_R(C')| - |\mathbb{D}_R(C')| - |\mathbb{E}_R(C')| = (k + y - 1) + (sum_R(C) - y) - |\mathbb{A}_R(C)| - (|\mathbb{D}_R(C)| - 1) - |\mathbb{E}_R(C')| = |\mathbb{B}_R(C)| + |\mathbb{C}_R(C)| = |\mathbb{B}_R(C')| + |\mathbb{C}_R(C')|.$

Case 1-2: Suppose line R3 of Remove($Z$) is performed with $mypred = &R$ between $C$ and $C'$. By Observation 8.10, $myprc - 1$ is added to $R.drc$, $|\mathbb{A}_R(C')| = |\mathbb{A}_R(C)|$, $|\mathbb{B}_R(C')| = |\mathbb{B}_R(C)|$, $|\mathbb{C}_R(C')| = |\mathbb{C}_R(C)|$, $|\mathbb{D}_R(C')| = |\mathbb{D}_R(C)|$, and $|\mathbb{E}_R(C')| = |\mathbb{E}_R(C)| - 1$. Since $Z \notin \mathbb{A}_R(C') \cup \mathbb{D}_R(C') \cup \mathbb{E}_R(C')$, $sum_R(C') = sum_R(C) - y$. Hence, in $C'$, $R.drc + sum_R(C') - |\mathbb{A}_R(C')| - |\mathbb{D}_R(C')| - |\mathbb{E}_R(C')| = (k + y - 1) + (sum_R(C) - y) - |\mathbb{A}_R(C)| - |\mathbb{D}_R(C)| - (|\mathbb{E}_R(C)| - 1) = |\mathbb{B}_R(C)| + |\mathbb{C}_R(C)| = |\mathbb{B}_R(C')| + |\mathbb{C}_R(C')|.$

Now consider the cases when $Z.prc$ is changed between $C$ and $C'$. $Z.prc$ is changed by line U5 of Update($Z$), line R2 of Remove($Z$), and line U3 with $mypred = &Z$.

Case 2: Suppose there exists some record $Z$ such that line U5 of Update($Z$) is performed between $C$ and $C'$.

Case 2-1: Suppose that line U5 of Update($Z$) is performed with $ppred = &R$ between $C$ and $C'$. By Observation 8.6, $Z.prc$ is set to 0, $|\mathbb{A}_R(C')| = |\mathbb{A}_R(C)| + 1$, $|\mathbb{B}_R(C')| = |\mathbb{B}_R(C)|$, $|\mathbb{C}_R(C')| = |\mathbb{C}_R(C)| - 1$, $|\mathbb{D}_R(C')| = |\mathbb{D}_R(C)|$, and $|\mathbb{E}_R(C')| = |\mathbb{E}_R(C)|$. Note that $sum_R(C') = sum_R(C)$ since $Z.prc = 0$. $R.rc$ is not changed, either. Therefore, in $C'$, $R.drc + sum_R(C') - |\mathbb{A}_R(C')| - |\mathbb{D}_R(C')| - |\mathbb{E}_R(C')| = k + sum_R(C) - (|\mathbb{A}_R(C)| + 1) - |\mathbb{D}_R(C)| - |\mathbb{E}_R(C)| = |\mathbb{B}_R(C)| + |\mathbb{C}_R(C)| - 1 = |\mathbb{B}_R(C')| + |\mathbb{C}_R(C')|.$
Case 2-2: Suppose that line U5 of Update(Z) is performed with \(Z.pred = (\&_R, x)\) between \(C\) and \(C'\). By Observation 8.7, \(Z.prc\) is set to 0, \(|A_R(C')| = |A_R(C)| -1, |B_R(C')| = |B_R(C)|, |C_R(C')| = |C_R(C)|, |D_R(C')| = |D_R(C)| + 1, and \(|E_R(C')| = |E_R(C)|\). Note that \(\text{sum}_R(C') = \text{sum}_R(C)\), since \(Z \in A_R(C), Z \in D_R(C')\), and \(Z.prc\) in \(C = \text{myprc}\) in \(C'\). \(R.rc\) is not changed, either. Therefore, in \(C'\), \(R.drc + \text{sum}_R(C') - |A_R(C')| - |D_R(C')| - |E_R(C')| = k + \text{sum}_R(C) - (|A_R(C)| - 1) - (|D_R(C)| + 1) - |E_R(C)| = |B_R(C)| + |C_R(C')| = |B_R(C')| + |C_R(C')|\).

Case 3: Suppose line R2 of Remove(Z) is performed with \(Z.pred = (\&_R, x)\) between \(C\) and \(C'\). By Observation 8.9, \(Z.prc\) is set to 0, \(|A_R(C')| = |A_R(C)| - 1, |B_R(C')| = |B_R(C)|, |C_R(C')| = |C_R(C)|, |D_R(C')| = |D_R(C)|, and \(|E_R(C')| = |E_R(C)| + 1\). Note that \(\text{sum}_R(C') = \text{sum}_R(C)\), since \(Z \in A_R(C), Z \in E_R(C')\), and \(Z.prc\) in \(C = \text{myprc}\) in \(C'\). \(R.rc\) is not changed, either. Therefore, in \(C'\), \(R.drc + \text{sum}_R(C') - |A_R(C')| - |D_R(C')| - |E_R(C')| = k + \text{sum}_R(C) - (|A_R(C)| - 1) - |D_R(C)| - (|E_R(C)| + 1) = |B_R(C)| + |C_R(C')| = |B_R(C')| + |C_R(C')|\).

Case 4: Suppose there exists some process \(W\) such that line U3 of Update(W) is performed with \(\text{myprc} = \&_Z\) between \(C\) and \(C'\) and \(Z.pred\) points to \(R\) in \(C\). Then, \(Z \in A_R(C)\). Since \(Z.predptr\) is not changed between \(C\) and \(C'\), \(Z \in A_R(C')\). This event sets \(ppred\) of \(W\) to the value of \(Z.predptr\), which is \(\&_R\). Hence, \(W\) is at line U4 with \(ppred = \&_R\), so \(W \in B_R(C')\). Thus, \(|B_R(C')| = |B_R(C)| + 1\). \(A_R, C_R, D_R, and E_R\) are not changed between \(C\) and \(C'\). \(Z.prc\) is incremented, so \(\text{sum}_R(C') = \text{sum}_R(C) + 1\). \(R.rc\) is not changed. Therefore, in \(C'\), \(R.drc + \text{sum}_R(C') - |A_R(C')| - |D_R(C')| - |E_R(C')| = k + (\text{sum}_R(C) + 1) - |A_R(C)| - |D_R(C)| - |E_R(C)| = |B_R(C)| + |C_R(C')| + 1 = |B_R(C')| + |C_R(C')|\).

The case where \(|A_R|\) is incremented is covered in Case 2-1. The case where \(|A_R|\) is decremented is covered in Case 2-2 and Case 3. The case where \(|B_R|\) is incremented is covered in Case 4. Whenever \(|B_R|\) is decremented, \(|C_R|\) is incremented by Observation 8.5, so the invariant still holds. The case where \(|C_R|\) is decremented is covered in Case
The case where $|D_R|$ is incremented is covered in Case 2-2. The case where $|D_R|$ is decremented is covered in Case 1-1. The case where $|E_R|$ is incremented is covered in Case 3. The case where $|E_R|$ is decremented is covered in Case 1-2.

**Invariant 8.13.** In any configuration $C$, if $R$ is live, Tail does not point to $R$, and $R.rc = (0, 0)$ in $C$, then $A_R(C) = B_R(C) = C_R(C) = D_R(C) = E_R(C) = \emptyset$.

*Proof.* Suppose that $R$ is live, Tail does not point to $R$, and $R.rc = (0, 0)$ in $C$. Since $R.orc = 0$, Invariant 8.11 implies $A_R(C) = C_R(C) = D_R(C) = E_R(C) = \emptyset$. Since $R.drc = 0$ and $A_R(C) = D_R(C) = E_R(C) = \emptyset$, Invariant 8.12 implies $B_R(C) = C_R(C) = \emptyset$. Therefore, $A_R(C) = B_R(C) = C_R(C) = D_R(C) = E_R(C) = \emptyset$.

**Lemma 8.14.** Let $C'$ be the configuration immediately after $C$. If $R$ is live, Tail does not point to $R$, and $R.rc = (0, 0)$ in $C$, then $R.rc = (0, 0)$ in $C'$ or $R$ is freed between $C$ and $C'$.

*Proof.* Since $R$ is live and Tail does not point to $R$, $R.rc$ can be updated only when some process performs line U4 with $ppred = \&R$, line U6 with $mypred = \&R$, or line R3 with $mypred = \&R$. By Invariant 8.13, $A_R(C) = B_R(C) = C_R(C) = D_R(C) = E_R(C) = \emptyset$. Since $B_R(C) = \emptyset$, no process performs line U4 with $ppred = \&R$ between $C$ and $C'$. Since $D_R(C) = \emptyset$, no process performs line U6 with $mypred = \&R$ between $C$ and $C'$. Since $E_R(C) = \emptyset$, no process performs line R3 with $mypred = \&R$ between $C$ and $C'$. Therefore, $R.rc = (0, 0)$ in $C'$ unless $R$ is freed by line R4 of Remove($R$) between $C$ and $C'$.

Invariant 8.13 and Lemma 8.14 imply the following lemma, which shows that, after $R.rc$ becomes $(0, 0)$, no process will access $R$ before $R$ is freed.

**Lemma 8.15.** If $R$ is live, Tail does not point to $R$, and $R.rc = (0, 0)$ in $C$, then $A_R(C') = B_R(C') = C_R(C') = D_R(C') = E_R(C') = \emptyset$ for all configurations $C'$ after $C$ but before $R$ is freed.
Lemma 8.16. Remove($R$) is called at most twice between when $R$ is allocated and $R$ is freed.

Proof. Remove($R$) is called from line D3 of Delete($R$), line U8 with $mypred = \&R$, and line R6 with $mypred = \&R$. By assumption, Delete($R$) is performed at most once, so line D3 of Delete($R$) is also performed at most once.

Suppose some process $p$ performs line U8 or R6 with $mypred = \&R$. Then, $(x, y) = (1, 1 − myprc)$ when $p$ last performed line U7 or R5, and $R.rc = (1, 1 − myprc)$ immediately before $p$ last performed line U6 or R3, respectively. Since $p$ added $(-1, myprc − 1)$ to $R.rc$ in lines U6 and R3, $R.rc = (0, 0)$ immediately after $p$ last performed line U6 and R3.

Let $p$ be the first process that performed line U6 or R3 with $mypred = \&R$ and $R.rc = (0, 0)$ after $R$ was allocated, and let $C$ be the resulting configuration. Then, by Lemma 8.15, $A_R(C') = B_R(C') = C_R(C') = D_R(C') = E_R(C') = \emptyset$ in any configuration $C'$ after $C$ but before $R$ is freed. Hence, no other process can perform line U6 or R3 with $mypred = \&R$ between $C$ and when $R$ is freed. Therefore, $p$ is the only process that performs line U8 or R6 with $mypred = \&R$.

Lemma 8.17. Only one process frees $R$.

Proof. By Lemma 8.16, Remove($R$) is only called twice. By line R1, only the second call of Remove($R$) can perform line R4.

Lemma 8.18. After $R$ is freed in line 4 of Remove($R$), no process will access $R$.

Proof. Lemma 8.15 shows that, if $R$ is live, $Tail$ does not point to $R$, and $R.rc = (0, 0)$, then no process will access $R$ before $R$ is freed. If $Tail$ points to $R$, then $R.rc$ was set to $(1, 1)$ in line A1, so no process performs line R2 of Remove($R$) before another record is Appended and the address of $R$ is read from $Tail$. Also, Lemma 8.17 shows that, if $R$ is freed, then no process frees $R$ again. Thus, no process will access a record after it is freed.
The above lemma shows that a record is not freed when a process may access it. Hence, we have the following theorem.

**Theorem 9.** The algorithm in Figure 9 is a correct wait-free implementation of S-HAD.

**Proof.** Note that $R.pred$ in Chapter 7 is called $R.predptr$ in Figure 9. All added lines and modifications to the algorithm in Chapter 7 deal with $rc$ and $prc$ and calling Remove. These events do not change $R.predptr$, $R.del$ or $Tail$, except for lines R2 and R4 of Remove($R$), in which $R.predptr$ is set to NIL and freed. By Lemma 8.17, only one process performs these lines. Note that, by lines U7 and R5, some process other than $owner(R)$ can call Remove($R$) only when $R.rc = (0, 0)$. By Lemma 8.15, no process can access $R$ after $R.rc$ is set to $(0, 0)$. Thus, when these lines are performed, no other process will access $R$.

Thus, in any configuration, all shared variables in the algorithm in Chapter 7 have the same values as the corresponding shared variables in the algorithm in Figure 9. Therefore, all observations, lemmas, invariant and corollaries in Chapter 7 also hold for the algorithm in Figure 9. Hence, the algorithm in Figure 9 correctly implements S-HAD.

---

### 8.4 Complexity analysis

In this section, we prove that this implementation of S-HAD uses $O(N^2)$ space. We also show that it has the same RMA complexity as the algorithm in Chapter 7. Therefore, the abortable mutual exclusion algorithm in Figure 7 using this implementation of S-HAD is local-spin, uses $O(N^2)$ space, has $O(N^2)$ RMA complexity and each process performs $O(k^2)$ RMAs in any passage, where $k$ is the number of processes that began the trying protocol immediately beforehand and subsequently aborted.
Space complexity of the algorithm

In this section, we show that $O(N^2)$ is an upper bound on the number of records in any configuration. This upper bound is achieved using our reference counting method, which frees a record $R$ when some process finds that $R.rc$ is $(0, 0)$.

The configuration graph of the algorithm in Chapter 7 is a tree rooted at the dummy record. The configuration graph of the algorithm in this chapter is a forest, consisting of a tree rooted at the dummy record and isolated nodes that represent records $R$ that have not been freed and such that $R.predptr$ was set to NIL by line R2 of Remove($R$).

The following lemma gives an upper bound on the maximum length of a path from a leaf to the dummy record.

**Lemma 8.19.** Consider any configuration $C$. Let $P$ be any path from a leaf to the root in $G(C)$. If $P$ has $k$ active records, then the length of $P$ is at most $(k+1)(2N-k)/2 + k$.

**Proof.** Let $W_1, \ldots, W_k$ be the active records on path $P$ such that $W_1 \prec W_2 \prec \ldots \prec W_k$. Let $W_0$ be the dummy node, and if $W_k$ is not a leaf, let $W_{k+1}$ be the inactive leaf on $P$. Note that the owners of $W_1, \ldots, W_i$ cannot own a record $Z$, for $W_i \prec Z$. Hence, for records $Z$ on the path such that $W_i \prec Z \prec W_{i+1}$, there are at most $N-i$ different owners for $Z$. By Invariant 7.15, the owners of those records $Z$ are all distinct. Hence, there are at most $N-i$ inactive records between $W_i$ and $W_{i+1}$ on the path, for $0 \leq i < k$. Between $W_k$ and $W_{k+1}$, there are $N-k-1$ inactive records since $W_{k+1}$ is also inactive. Hence, the length of $P$ is at most $N+(N-1)+(N-2)+\ldots+(N-k)+k = (k+1)(2N-k)/2 + k$. $\Box$

The following lemma shows that the number of leaves in the tree is bounded above by $2N$. Note that the configuration tree in Chapter 7 has unbounded degree.

**Lemma 8.20.** In $G(C)$, there are at most $N$ active leaves and $N$ inactive leaves in the subtree rooted at the dummy record and at most $N$ isolated nodes.

**Proof.** If the record at the end of the sequence is inactive in $C$, then the owner of the record does not own any active record in $C$. In this case, there are at most $N-1$ active
leaves in $C$. If the record at the end of the sequence is active in $C$, then there are at most $N$ active leaves in $G(C)$.

Let $R$ be an inactive leaf in the subtree rooted at the dummy record in $G(C)$, and suppose $Tail$ does not point to $R$. If $R.orc \neq 0$ in $C$, then $A_R(C) \cup C_R(C) \cup D_R(C) \cup E_R(C) \neq \emptyset$, by Invariant 8.11. Since there is no record that points to $R$ in $G(C)$, $A_R(C) = \emptyset$, so $C_R(C) \cup D_R(C) \cup E_R(C) \neq \emptyset$. Thus, there exists an active record $Z \in C_R(C) \cup D_R(C) \cup E_R(C)$.

If $R.orc = 0$ and $R.drc \neq 0$, then by Invariant 8.11, $A_R(C) = C_R(C) = D_R(C) = E_R(C) = \emptyset$. Thus, $sum_R(C) = 0$ and by Invariant 8.12, $|B_R(C)| = R.drc$. Thus, there exists an active record in $B_R(C)$.

Now consider when $R.rc = (0,0)$. Since $R$ is inactive in $C$, $owner(R)$ has finished performing $Delete(R)$. Since $R$ has not been freed, $Remove(R)$ called from $Delete(R)$ did not free $R$. Hence, when line R1 of $Remove(R)$ called from $Delete(R)$ was performed, $test\_and\_set(R.done)$ returned 0 and $R.done$ was set to 1. Since $R.rc = (0,0)$, there was a process $p$ that set $R.rc$ to $(0,0)$ in line U6 or R3 with $mypred = &R$.

If $p$ keeps performing steps, then its test on line U7 or R5 will be successful, and it will call $Remove(R)$. In line R1 of $Remove(R)$, the $test\_and\_set(R.done)$ will returns 1 and $p$ will perform lines R2 and R3 before $R$ is freed on line R4. However, when $p$ performs line R2 of $Remove(R)$, $R.predptr$ is set to NIL, so $R$ becomes an isolated node. In $C$, $R$ is not an isolated node, so $p$ has not yet performed line R2 of $Remove(R)$. Thus, $p$ is either at line U7, U8, R1 of $Remove(R)$, R2 of $Remove(R)$, R4 of $Remove(Z)$, R5 of $Remove(Z)$, or R6 of $Remove(Z)$, for some record $Z$. Hence, $p$ cannot be at lines U4, U5, U6 or R3. Let $W$ be the active record owned by $p$ in $C$. Then, $W \notin B_{R'}(C) \cup C_{R'}(C) \cup D_{R'}(C) \cup E_{R'}(C)$ for any record $R'$.

Note that, if $Z \in B_{R'}(C) \cup C_R(C) \cup D_R(C) \cup E_R(C)$, then $Z \notin B_{R'}(C) \cup C_{R'}(C) \cup D_{R'}(C) \cup E_{R'}(C)$ for any $R' \neq R$. Therefore, for each inactive leaf $R$ in the subtree rooted at the dummy record in, except for possibly the leaf $Tail$ points to, there exists
a different active record. Since there are at most \( N \) active records, there are at most \( N \) inactive leaves in \( G(C) \).

If a process is at line R3 or R4 of Remove(\( R \)), then \( R\text{.predptr} \) is set to NIL, but \( R \) has not been freed. Hence, \( R \) is an isolated node in \( G(C) \). Thus, for each isolated node in \( G(C) \), there exists a different active process. Therefore, there are at most \( N \) isolated node in \( G(C) \).

**Lemma 8.21.** In any configuration \( C \), each process owns at most \( 3N - 1 \) records in \( G(C) \).

**Proof.** In \( G(C) \), there is at most one active record owned by each record. Let \( X \) and \( Y \) be the inactive records owned by the same process, \( p \), and suppose that \( Y \) is an ancestor of \( X \) in \( G(C) \). Then, by Invariant 7.15, there exists an active record on the path from \( X \) to \( Y \) in \( G(C) \). By Lemma 8.20, there are at most \( N \) active leaves, \( N \) inactive leaves, and \( N \) isolated nodes in \( G(C) \). Since \( X \) and \( Y \) are on the same path in \( G(C) \), they cannot be at the isolated node in \( G(C) \). Since there exists an active record on the path from \( X \) to \( Y \) in \( G(C) \), if there are \( k \) pairs of such \( X \) and \( Y \), there are at most \( N - k \) active leaves in \( G(C) \). Hence, there are at most \( N - 1 \) pairs of such \( X \) and \( Y \) in \( G(C) \). Therefore, there are at most \( 3N - 2 \) inactive records (\( 2N - 2 \) for such pairs like \( X \) and \( Y \), and \( N \) for isolated nodes) in \( G(C) \). In total, there are at most \( 3N - 1 \) records owned by \( p \) in \( G(C) \). \( \square \)

Lemma 8.21 shows that, if we use a free list, then it is enough for each process to have \( 3N - 1 \) different records. However, there are actually fewer records in \( G(C) \).

**Lemma 8.22.** In any configuration \( C \), there are at most \( 2N^2 + 2N \) records in \( G(C) \).

**Proof.** Let \( M(k, i) \) be the maximum number of records of a tree rooted at the dummy record in \( G(C) \) with \( k \) active records and \( i \) leaves, and let \( T(k, i) \) be the tree with \( M(k, i) \) records. If a path from a leaf to the root does not contain an active record, the length of
the path is at most \( N \) by Lemma 8.19. Hence, \( M(0, N) = N^2 \) and \( T(0, N) \) is a tree with \( N \) branches at the root.

Note that, by Lemma 8.19, the difference of the maximum lengths of the path with \( k \) active records and the path with \( k - 1 \) active records is \( N - k + 1 \). Hence, when we add an active record to a path with less active records, we can increase the maximum number of the path more. Now we add \( N \) active records one by one to \( T(0, N) \) to create \( T(N, N) \).

Note that, for \( 0 \leq i < N \), there still exists a branch without an active record in \( T(i, N) \), since \( T(0, N) \) has \( N \) different branches at the root. Hence, we can build \( T(i + 1, N) \) from \( T(i, N) \) by adding an active record to one of those branches that does not contain any active record. This increases the maximum length of the branch from \( N \) to \( 2N \) by Lemma 8.19. Hence, \( M(i + 1, N) = M(i, N) + N \), so \( M(k, N) = N^2 + kN \).

By Lemma 8.20, there are at most \( 2N \) leaves and at most \( N \) isolated nodes in \( G(C) \). However, among those \( 2N \) leaves, at most \( N \) are active and at most \( N \) are inactive. The maximum length of the path from an active leaf to a root that does not contain an additional active record is \( N + 1 \). Thus, if there are \( i \) active leaves and \( N - i \) active records are at the inner nodes, then the maximum number of records in the tree is 
\[
i(N + 1) + M(N - i, N) = i(N + 1) + N^2 + (N - i)N = i + 2N^2.
\] Since \( i \) is bounded by \( N \), the maximum number of records in the tree is \( 2N^2 + N \). So, the maximum number of records in \( G(C) \) is \( 2N^2 + 2N \).

\[\square\]

**Corollary 8.23.** In any configuration \( C \), if there are only \( k \) processes that have invoked the algorithm, then there are at most \( 2k^2 + 2k \) records in \( G(C) \).

**Proof.** Lemma 8.22 holds when there are \( N \) processes. If only \( k \) processes have invoked the algorithm, then the whole execution can be regarded as when there are \( k \) processes in total. Hence, there are at most \( 2k^2 + 2k \) records in \( G(C) \).

\[\square\]

Lemma 8.22 implies the following theorem.
Theorem 10. The abortable mutual exclusion algorithm in Chapter 6 from S-HAD implemented in Figure 9 uses $\Theta(N^2)$ space.

Worst case RMA analysis of the algorithm

We prove that $O(k^2)$ RMAs are performed during any call of Remove, where $k$ is the number of processes that began the trying protocol immediately beforehand and subsequently aborted. Since the algorithm in Chapter 7 also has $O(k^2)$ RMA complexity, the algorithm in Chapter 8 has the same RMA complexity.

Invariant 8.24. During Update($R$) or Remove($R$), if mypred = &S or ppred = &S, then $S \prec R$.

Proof. If mypred = &S during Update($R$) or Remove($R$), then mypred is assigned from $R$.predptr, so $R$.predptr = &S. Then, by Invariant 7.7, $S \prec R$. If ppred = &S during Update($R$), then ppred is assigned from (*mypred).predptr, so (*mypred).predptr = &S.

Let mypred = &T, then T.predptr points to S. Thus, by Invariant 7.7, $S \prec T$. Also, since mypred = &T, $T \prec R$. Therefore, $S \prec R$.

Invariant 8.25. If owner($R$) performs Remove($S$) and $R \neq S$, then $S \prec R$.

Proof. Since $R \neq S$, Remove($S$) is called from line U8 of Update($R$) with mypred = &S or line R6 of Remove($Z$) with mypred = &S, where $Z$ is some record. If line U8 of Update($R$) with mypred = &S calls Remove($S$), then by Invariant 8.24, $S \prec R$. Otherwise, we assume that, if owner($R$) recursively calls Remove $k$ times and performs Remove(*mypred) with mypred = &$Z_k$, then $Z_k \leq R$. When $k = 0$, owner($R$) directly calls Remove($Z_0$) from either line D3 of Delete($R$) or line U8 of Update($R$), so $Z_0 \leq R$. If owner($R$) recursively calls Remove $k+1$ times and performs Remove($S$), then it performs line R6 of Remove($Z_k$) with mypred = &S, so by Invariant 8.24, $S \prec Z_k$. Hence, $S \prec R$. Therefore, in any case, $S \prec R$. 

\(\square\)
Lemma 8.26. Let \( R' \) be the last record that was appended prior to element \( R \), but was not prematurely deleted. Let \( C \) be the configuration when \( \text{Append}(R) \) was invoked. If \( k \) is the number of different processes that appended records between \( R' \) and \( R \) inclusive, then there are \( 2k^2 + 2k \) records in the subtree rooted at \( R' \) in \( G(C) \).

Proof. Since \( R' \) was not prematurely deleted, \( R'.\text{del} = \text{false} \) before \( R'.\text{pred} \) points to the dummy record. By Invariants 7.7, for any records \( X \) and \( Y \) such that \( X \prec R' \) and \( R' \preceq Y \), \( X.\text{pred} \) does not point to \( Y \) in any configuration. Hence, all records in the subtree rooted at \( R' \) were appended after \( R' \). Since only \( k \) processes have invoked the algorithm between when \( R' \) was appended and \( C \), this implies that all records in the subtree rooted at \( R' \) in \( G(C) \) were appended by the \( k \) processes. Hence, the maximum number of records in the subtree rooted at \( R' \) in \( G(C) \) is the same as the maximum number of records in the configuration tree when only \( k \) processes have invoked the algorithm, which is \( 2k^2 + 2k \) by Corollary 8.23. \( \square \)

Lemma 8.27. Let \( R' \) be the last record that was appended prior to element \( R \), but was not prematurely deleted. If \( k \) is the number of different processes that appended records between \( R' \) and \( R \) inclusive, then \( O(k^2) \) RMAs are performed by \( \text{Update}(R) \) and \( \text{Remove}(R) \) between beginning \( \text{Append}(R) \) and completing \( \text{Delete}(R) \).

Proof. By Lemma 7.17, \( \text{Update} \) generates \( O(k^2) \) RMAs. Let \( C \) be the configuration when \( \text{Append}(R) \) is invoked. By Invariant 8.25, if \( \text{owner}(R) \) performs \( \text{Remove}(S) \) and \( S \neq R \), then \( S \prec R \). By Invariant 7.11, \( R.\text{pred} \) does not point to any record \( X \), where \( X \prec R' \), before \( R' \) is logically deleted from the head. Hence, if \( \text{owner}(R) \) performs \( \text{Remove}(S) \) before \( R' \) is deleted from the head, then \( R' \prec S \prec R \). Thus, \( S \) is in the subtree rooted at \( R' \) in \( G(C) \). By Lemma 8.26, there are \( O(k^2) \) records in the subtree rooted at \( R' \) in \( G(C) \). When \( R \) is logically deleted, \( R.\text{pred} \) points to the dummy record. Hence, \( \text{owner}(R) \) cannot call \( \text{Remove}(T) \) for any \( T \prec R' \) even after \( R' \) is logically deleted. Therefore, \( \text{Remove}(R) \) is called \( O(k^2) \) times. \( \square \)
Since $k$ is bounded by $N$, Lemma 7.18 and Lemma 8.27 imply the following theorem.

**Theorem 11.** The worst case RMA complexity from the beginning of Append($R$) to the end of Delete($R$) in the S-HAD implementation in Figure 9 is $\Theta(N^2)$.

By Theorem 9, the algorithm in Figure 9 is a correct implementation of S-HAD. Hence, the abortable mutual exclusion algorithm using this implementation of S-HAD is correct. By Theorem 10, this algorithm uses only $\Theta(N^2)$ shared variables. By Theorem 11, the worst case RMA complexity of the abortable mutual exclusion algorithm is $\Theta(N^2)$. Also, Lemma 8.27 shows that each process performs $\Theta(k^2)$ RMAs in any CC model, where $k$ is the number of processes that immediately began the trying protocol immediately beforehand and subsequently aborted. Hence, when no abort occurs, this algorithm has $O(1)$ RMA complexity.

**Theorem 12.** There exists a queue-based local-spin abortable mutual exclusion algorithm that uses $\Theta(N^2)$ shared variables, READ, WRITE, FETCH\_AND\_STORE, FETCH\_AND\_ADD and TEST\_AND\_SET operations, satisfies mutual exclusion, lockout freedom, bounded exit and bounded abort, and has $\Theta(k^2)$ RMA complexity in the CC model, where $k$ is the number of processes that immediately began the trying protocol immediately beforehand and subsequently aborted.
Chapter 9

Open Problems

There are several open problems arising from our results. First, our algorithm in Chapter 8 is local-spin only for the CC model. The existence of a local-spin abortable mutual exclusion algorithm with $O(1)$ RMA complexity in the absence of abort using bounded space on the DSM model is still open. Also, the space and RMA complexity of our algorithm in Chapter 8 is $O(N^2)$, which is higher than the space and RMA complexity of many local-spin mutual exclusion algorithms. Hence, it is open whether $\Omega(N^2)$ space and RMAs are necessary for local-spin abortable mutual exclusion algorithms with $O(1)$ RMA complexity in the absence of abort in the CC model. All algorithms in this thesis and all existing abortable mutual exclusion algorithms use atomic operations. It is open whether there exists a local-spin abortable mutual exclusion algorithm using non-atomic registers.

When there are $L$ locks, the space complexity of our algorithms increase by a factor of $L$. In mutual exclusion, there exists an algorithm that only needs additional $O(L)$ space when there are $L$ locks [MCS91, Cra93]. Scott [Sco02] conjectured it is impossible to have an abortable mutual exclusion algorithm with $O(L + S)$ space where $L$ is the number of locks and $S$ is the space complexity of the algorithm with one lock. This question remains open.
Our abortable mutual exclusion algorithms satisfy lockout freedom. However, in some situations, deadlock freedom is sufficient. For example, a \texttt{test\_and\_set}-based try-lock, which is described in Chapter 1, is widely used in practice [MCS91]. It is an abortable mutual exclusion algorithm that is not local-spin and does not satisfy lockout freedom. It is unknown whether there exists a local-spin abortable mutual exclusion algorithm that does not satisfy lockout freedom and has better space and RMA complexity than existing local-spin abortable mutual exclusion algorithms.

In this thesis, we only considered atomic operations. Kim and Anderson [KA06] presented a local-spin mutual exclusion algorithm using non-atomic \texttt{read} and \texttt{write}. However, if we apply the transformation in Chapter 4 to this algorithm, the resulting algorithm is no longer correct. It is open whether there exists a local-spin abortable mutual exclusion algorithm using non-atomic \texttt{read} and \texttt{write}.

In this thesis, we considered only system models without process failures. In mutual exclusion or abortable mutual exclusion, a process crash in the critical section prohibits other processes from entering the critical section. However, it is open whether there exists an abortable mutual exclusion algorithm that tolerates a crash during the trying, exit and abort protocols. Some existing \textit{k}-exclusion algorithms tolerate up to \(k - 1\) crash failures. It is open whether there exists an abortable \textit{k}-exclusion algorithm that tolerates \(k - 1\) crash failures. In abortable mutual exclusion, a process can only abort its own passage. However, if a slow process starts waiting before a fast process, the fast process may want to abort the passage by the slow process. It is open whether there exists an algorithm for this variant of abortable mutual exclusion.
Bibliography


[Dan10] Robert Danek. The k-Bakery: Local-spin k-Exclusion Using Non-atomic Reads and Writes. In *Proceeding of the 29th ACM SIGACT-SIGOPS sym-


Index of Definitions

abortable mutual exclusion problem, 17
ATOMIC_ADD, 8
bounded abort, 17
bounded exit, 15
busy-waiting, 20
cache-coherent (CC) model, 10
with cache-invalidate protocol, 12
with cache-update protocol, 11
COMPAREAndPassword_SWAP (CAS), 7
comparison-based operation, 7
concurrent entering, 19
configuration, 8
configuration graph, 103
crash failure, 18
deadlock freedom, 15
k-deadlock freedom, 18
deallocation, 42
distributed reference counter, 112
distributed shared memory (DSM) model, 10
doorway, 16
dormant, 77
dynamic node, 42
event, 8
k-exclusion problem, 18
execution, 10
execution fragment, 8
FETCHAndPassword_ADD, 8
FETCHAndPassword_INC, 8
FETCHAndPassword_STORE (SWAP), 8
first-come first-served (FCFS), 16
free-list, 42
garbage, 41
group mutual exclusion, 19
group mutual exclusion problem, 19
LOAD-LINKED/STORE-CONDITIONAL (LL/sc), 7
local pointer, 42
local-spin, 20
lockout freedom, 15
memory allocation, 41
memory reclamation, 42
mutual exclusion, 15
mutual exclusion problem, 15
node, 41
original reference counter, 112
passage, 15
pointer, 41
premature reclamation, 43
proactive reference counter, 112
queue-based, 16
reachable configuration, 10
reachable node, 42
reference, 42
S-HAD, 90
shared pointer, 42
spin variable, 20
state of a process, 8
static node, 42
tagging, 7
TEST_AND_SET, 7
try-lock, 2
trying process, 15
unreachable node, 42