LOCAL-SPIN ALGORITHMS FOR VARIANTS OF MUTUAL EXCLUSION USING READ AND WRITE OPERATIONS

by

Robert Danek

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

Abstract

LOCAL-SPIN ALGORITHMS FOR VARIANTS OF MUTUAL EXCLUSION USING READ AND WRITE OPERATIONS

Robert Danek

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2011

Mutual exclusion (ME) is used to coordinate access to shared resources by concurrent processes. We investigate several new N-process shared-memory algorithms for variants of ME, each of which uses only reads and writes, and is local-spin, i.e., has bounded remote memory reference (RMR) complexity. We study these algorithms under two different shared-memory models: the distributed shared-memory (DSM) model, and the cache-coherent (CC) model. In particular, we present the first known algorithm for first-come-first-served (FCFS) ME that has $O(\log N)$ RMR complexity in both the DSM and CC models, and uses only atomic reads and writes. Our algorithm is also adaptive to point contention, i.e., the number of processes that are simultaneously active during a passage by some process. More precisely, the number of RMRs a process makes per passage in our algorithm is $\Theta(\min(c, \log N))$, where c is the point contention. We also present the first known FCFS abortable ME algorithm that is local-spin and uses only atomic reads and writes. This algorithm has $O(N)$ RMR complexity in both the DSM and CC models, and is in the form of a transformation from abortable ME to FCFS abortable ME. In conjunction with other results, this transformation also yields the first known local-spin group mutual exclusion algorithm that uses only atomic reads and writes. Additionally, we present the first known local-spin k-exclusion algorithms that use only atomic reads and writes and tolerate up to $k - 1$ crash failures. These

algorithms have RMR complexity $O(N)$ in both the DSM and CC models. The simplest of these algorithms satisfies a new fairness property, called k-FCFS, that generalizes the FCFS fairness property for ME algorithms. A modification of this algorithm satisfies the stronger first-in-first-enabled (FIFE) fairness property. Finally, we present a modification to the FIFE k-exclusion algorithm that works with non-atomic reads and writes. The high-level structure of all our k-exclusion algorithms is inspired by Lamport's famous Bakery algorithm.

Acknowledgements

Firstly I would like to thank my supervisor, Vassos Hadzilacos, for the insight and guidance that he provided during the writing of this thesis. He provided a critical perspective that encouraged me to constantly improve my work. Most importantly, he taught me that solving a problem is only half of the challenge in doing a Ph.D. Being able to write up one's results as clearly as possible is just as difficult and important as solving a hard problem in the first place. As such, I've strived to provide a lucid exposition of all the results in this thesis. If the reader discovers any parts of this thesis in which complex ideas are conveyed with exceptional clarity, some of the credit must be given to Vassos for having encouraged me to write and rewrite until that level of clarity was reached.

I would also like to thank my other committee members, Sam Toueg, Angela Demke Brown, and my External Appraiser, Prasad Jayanti. They provided valuable feedback on my work and helped shape this thesis into its current form. In addition, I would like to thank Allan Borodin, who temporarily served on my committee, for his helpful comments.

I want to thank Faith Ellen for the time that she took to provide feedback on my joint work with Hyonho Lee in [15]. Faith's precision to detail helped to improve the quality of that work.

I want to thank my main collaborators, Hyonho Lee and Wojciech Golab. Hyonho's work on k-exclusion, which appears in [15, 35], inspired me to investigate the problem further and culminated in the results found in Chapter 5. Wojciech worked jointly with me on the $O(\log N)$ FCFS mutual exclusion algorithm that appears in Chapter 3. The ticket dispenser and *SpecialSet* components that are defined in that chapter are mostly his work, and appear in more detail in [13].

Besides collaborating with me on papers, I also want to thank Hyonho and Wojciech for the many insightful discussions that we had on various problems in theoretical distributed computing.

Lastly, I would like to thank my family for the support that they've given me. In particular, I would like to thank my wife, Melissa, for her love, encouragement, and her patience, while I completed my Ph.D. She always wanted to celebrate every small milestone I achieved on the path to receiving my degree, and she was always there to provide motivation during those times when making progress on my thesis seemed difficult. Finishing my Ph.D. without her support would have been a million times harder, and I'm very thankful to her for being there.

Contents

List of Algorithms

Chapter 1

Introduction

Shared-memory multiprocessor computers have a significant speed advantage over single processor computers, since they allow multiple processes to execute tasks concurrently. With this speedup, however, there arise resource coordination issues that must be handled. Some resources in a computer can only be accessed by one process at a time. For example, a hardware resource such as a printer can only print one job at a time, and a software resource such as a queue can only be modified by one process at a time. To solve this problem, a mutual exclusion (ME) algorithm can be used to protect the resource.

An algorithm for ME consists of a *trying protocol* (TP) and an *exit protocol* (EP). Before a process can access the shared resource (also known as the *critical section* (CS)), it executes the TP, and after it finishes the CS, it executes the EP. A process that is not in the TP, CS, or EP is said to be in the *non-critical section* (NCS).

A process may try accessing the CS multiple times throughout its lifetime. We illustrate this in Figure 1.1 using an infinite loop.

The TP and EP of a mutual exclusion algorithm must be written in such a way to satisfy the following properties:

Mutual Exclusion: No two processes are in the CS simultaneously.

Lockout Freedom: If some process p is in the TP, then p eventually enters the CS.

Figure 1.1: The execution cycle

loop

Non-Critical Section (NCS)

Trying Protocol (TP) { Doorway (DWY); Waiting (WRM) }

Critical Section (CS)

Exit Protocol (EP)

end loop

Bounded Exit: If a process p enters the EP, then p returns to the NCS in a bounded number of its own steps.

The description of bounded exit requires that a process leaves the exit protocol "in a bounded number of its own steps". The idea of a process 'step' is explained more fully in the next chapter where we define the model, but for now it can be thought of as a read or a write to a single shared variable. The intuition behind bounded exit is that a process can finish the exit protocol without having to wait for other processes. More precisely, whenever we say that a process p completes some section of code "in a bounded number of its own steps", we mean that there exists a function $f(N)$, dependent only on N, the number of processes in the system, such that p is guaranteed to have finished the segment of code after having executed $f(N)$ steps.

Lockout freedom is sometimes replaced with a weaker property called deadlock freedom:

Deadlock Freedom: If some process is in the TP, then some process eventually enters the CS.

There are several variants of the mutual exclusion problem. We introduce below the ones that are studied in this thesis. For each of these variants, we assume that the critical

section is finite, i.e., after a process p takes some finite (but unbounded) number of steps in the CS, p is guaranteed to leave the CS and start the EP. Also, with the exception of the k-exclusion variant, we assume that every process that leaves the NCS will keep taking steps at least until it returns to the NCS. That is, a process does not crash while it is outside of the NCS.

1.1 FCFS ME

The properties stated in the preceding section, which are satisfied by ordinary mutual exclusion algorithms, do not prevent situations in which a process waits inside the trying protocol for a long time while other processes are repeatedly granted entry to the critical section. Intuitively, this is unfair. To rectify this situation, we introduce a new property known as the First-Come-First-Served (FCFS) property [32], which informally requires that processes are granted entry into the CS in the order in which they leave the NCS. To make this more precise, the trying protocol is split into two parts (see Figure 1.1): the first part, the doorway (DWY), which a process completes in a bounded number of its own steps; and a second part, the waiting room (WRM). FCFS is defined as follows:

FCFS: If a process p finishes executing the doorway before a process q begins executing the doorway, then q does not enter the CS before p.

The TP and EP of an FCFS ME algorithm satisfy all of mutual exclusion, lockout freedom, bounded exit, and FCFS.

1.2 Abortable ME

If a process enters the trying protocol, it cannot leave the trying protocol until it is granted access to the critical section. A variant of mutual exclusion that lifts this restriction is *abortable* mutual exclusion [41]. In this variant, a process can withdraw its request to

enter the CS by executing a section of code known as the *abort protocol* (AP), and then return to the NCS. A process can only withdraw its request at certain points in the trying protocol, known as *abort points*. If a process leaves the trying protocol and starts executing the abort protocol anywhere else, one or more of the correctness properties of the ME algorithm may fail to hold.

Scott and Scherer [41] point out several applications where this functionality may be useful, including real-time systems, in which the time spent waiting for a resource cannot exceed a certain threshold, and database systems, which can use the functionality to recover from suspected deadlocks.

The TP and EP of an abortable ME algorithm must satisfy the same properties as in ordinary ME. The TP and AP must also satisfy the following property, which has two parts:

Bounded Abort: (i) If a process is in the trying protocol and cannot enter the CS in a bounded number of its own steps, then it can reach an abort point in a bounded number of its own steps; and (ii) If a process starts executing the abort protocol while at an abort point, then it returns to the NCS in a bounded number of its own steps.

This property ensures that when a process wants to abort its attempt to enter the CS, it can perform the abort without having to wait for other processes. Without this property, abortability becomes a feature of little practical value: The main purpose of abortability is to allow a process to return to the NCS if it decides that it has been waiting too long to access the CS. If, upon deciding to abort, a process still has to wait an unbounded amount of time, it might as well have continued to wait to enter the CS.

We also study in this thesis FCFS abortable ME. The FCFS property in this variant of ME is modified slightly to read: If a process p finishes executing the doorway before a process q begins executing the doorway, then q cannot enter the CS before p enters the CS or aborts its entry attempt.

1.3 Group ME

Group mutual exclusion [28] is similar to mutual exclusion, except that a process requests a "session" before leaving the NCS, and processes that request the same session can enter the CS concurrently. An algorithm for N-process group mutual exclusion (GME) consists of a trying and exit protocol that satisfy lockout freedom and bounded exit, as in "ordinary" mutual exclusion, plus the following properties:

Group Mutual Exclusion: No two processes requesting different sessions are in the CS simultaneously.

Concurrent Entering: If a process p is requesting a session s, and no other process is requesting a session $s' \neq s$, then p enters the CS in a bounded number of its own steps.

Mutual exclusion is a special case of group mutual exclusion where each process requests its own ID as its session every time it leaves the NCS, with one minor difference. In this special case, the concurrent entering property says that if a process p tries to enter the CS while all other processes are in the NCS, then p enters the CS in a bounded number of its own steps. Ordinary mutual exclusion does not require that this property be satisfied,[1] although all mutual exclusion algorithms of which we are aware do actually satisfy it. To not satisfy this property, an algorithm would likely have to do something strange such as choose a random number from an unbounded range and then wait that number of steps before attempting to enter the CS. Adding this property to the specification of mutual exclusion would not really make the problem any more difficult, but we omit it so as to remain consistent with the way mutual exclusion is traditionally defined.

One use for group mutual exclusion is in a Computer Supported Cooperative Work application, described by Joung [28]. Such an application may have an electronic white

[1] We are thankful to Sam Toueg for pointing out this observation.

board (the critical section), to which multiple users (the processes) have access. When a user wants to share some piece of information about a certain topic (the session) with other users, he posts that information to the white board. Users interested in the same topic may then access that information, possibly altering it or adding their own information. Users that are interested in a different topic, however, cannot use the white board until the current users are finished.

Another use for group mutual exclusion is in solving the Readers-Writers (RW) problem [11]. In the RW problem, readers are allowed concurrent access to the critical section, whereas each writer requires exclusive access. A GME algorithm can be used to solve the RW problem by having readers request a common session (e.g., session 0), while each writer requests a unique session (e.g., its own process ID, assuming process IDs are greater than 0).

1.4 k-Exclusion

k-Exclusion [19] is a generalization of mutual exclusion that allows up to k processes to be in the critical section concurrently, and can tolerate up to $k-1$ process *crashes* in a manner described more precisely in the starvation freedom property below. We say that a process crashes if it stops taking steps while outside the NCS. If a process crashes we say that it is *faulty*; otherwise we say that it is *non-faulty*. Note that, for our purposes, a process that stops taking steps in the NCS, i.e., a process that never attempts to enter the CS after some point, is not considered faulty.

The TP and EP of a k-exclusion algorithm must satisfy the following properties:

k-Exclusion: At most k processes are in the CS simultaneously.

Starvation Freedom: If a non-faulty process p is in the TP and at most $k-1$ other processes crash, then p eventually enters the CS.

A k-exclusion algorithm must additionally satisfy bounded exit, whose statement remains unchanged from the ordinary mutual exclusion problem. Note that mutual exclusion is a special case of k-exclusion where $k = 1$, except that starvation freedom is called lockout freedom in the context of ordinary mutual exclusion.

Certain k-exclusion algorithms may satisfy a weaker liveness property than starvation freedom, known as deadlock freedom:

Deadlock Freedom: If a non-faulty process is in the TP, and at most $k - 1$ other processes crash, then some process eventually enters the CS.

An equivalent formulation of the deadlock freedom property, which we use in this thesis, says that if a non-faulty process p is stuck in the trying protocol forever, and at most $k-1$ processes crash, then some process executes an infinite number of times through the CS. (This property should not be confused with the deadlock freedom property that we defined for ordinary mutual exclusion. When we make subsequent references to this property, it will be clear from the context which version we mean.)

There are also variants of k-exclusion in which the order that processes are admitted into the CS is more fair than is guaranteed by the properties above. For ordinary mutual exclusion, the FCFS property was used to ensure fair ordering. In the context of k-exclusion, when $k > 1$, requiring that an algorithm satisfy FCFS does not make sense, since it conflicts with the starvation freedom property: If a process p completes the doorway and crashes before a non-faulty process q enters the doorway, then a k-exclusion algorithm that satisfies FCFS must prevent q from ever entering the CS, thereby violating starvation freedom. However, we can weaken the FCFS property so that it does not conflict with starvation freedom and still provides some modicum of fairness. Our new property is *k-FCFS*:

k-FCFS: For any set of processes Y such that $|Y| = k$, if all processes in Y finish the doorway before a process p starts the doorway, then p does not enter the CS before

at least one process in Y enters the CS.

A nice feature of k-FCFS is that it makes sense for all $k \geq 1$. In particular, for $k = 1$, it is simply the FCFS property for ordinary mutual exclusion.

Another fairness property for the k-exclusion problem, known as *First-In-First-Enabled (FIFE)*, was originally defined by Fischer et al. [19]. Intuitively, this property captures the notion of fair ordering by requiring that processes become *enabled* to enter the CS in the order in which they execute through the doorway. A process p is enabled to enter the CS if it can enter the CS in a bounded number of its own steps. More precisely:

FIFE: If a process p finishes the doorway before a process q starts the doorway, and q enters the CS before p, then p can enter the CS in a bounded number of its own steps.

FIFE is a stronger property than k-FCFS in that any k-exclusion algorithm that satisfies FIFE also satisfies k-FCFS. To see why this is the case, suppose, by way of contradiction, that there is a k-exclusion algorithm that satisfies FIFE but not k-FCFS. There exists an execution of this algorithm in which some set of processes Y, where $|Y| = k$, all finish the doorway before a process p starts the doorway, and p enters the CS before any process in Y enters the CS. By FIFE, each process in Y is able to enter the CS in a bounded number of its own steps. Thus, the execution can proceed in such a way that process p remains in the CS until all processes in Y enter the CS. We now have a situation in which $k + 1$ processes are in the CS, contradicting that the algorithm satisfies k-exclusion.

1.5 Models and Complexity Measures

The message-passing model and the shared-memory model are two ways of modeling the communication between processes. In the message-passing model, processes communicate

by sending messages to each other through a "network", and in the shared-memory model processes communicate by reading and updating shared variables to which processes have access. In this thesis we focus on the shared-memory model.

The shared-memory model is further divided into the following two models [4]: the distributed shared-memory (DSM) model, and the cache-coherent (CC) model. These models provide a more precise description of how processes access shared variables.

In the DSM model, each process has a memory module associated with it. The shared variables used by the algorithm are partitioned among the different memory modules before an execution begins and this partitioning remains invariant during the algorithm's execution. Whenever a process accesses a shared variable stored in another process's memory module, a *remote memory reference* occurs. When a process accesses a shared variable in its own memory module, a *local memory reference* occurs.

In the CC model there is a "main memory", remote to each process, that all processes can access. Each process also has a local cache of unbounded size. Any time a process writes a shared variable, it writes it to main memory, hence making a remote memory reference. Whenever a process reads a shared variable, it first attempts to read a copy of the variable from its cache. If a copy does not exist in the process's cache, the process will read the variable from main memory, thereby incurring a remote memory reference, and then will cache a copy of the variable locally. A cached copy of the variable v remains in a process p's cache until the copy is *invalidated*, at which time it is effectively erased from p's cache. All cached copies of v are invalidated whenever any process writes v to main memory. There are other variants of the CC model, which we describe in more detail in Chapter 2.

A common approach for measuring the time complexity of an algorithm is to count the worst-case number of memory references that a process can make when it executes the algorithm. We refer to this as an algorithm's *step complexity*. This approach, however, is problematic for mutual exclusion algorithms, since a process can make an unbounded

number of memory references if it needs to wait to enter the CS. Instead of counting all memory references, we only count *remote* memory references (RMRs). More precisely, we will measure the efficiency of mutual exclusion algorithms using *RMR complexity* [4]: the maximum number of RMRs a process incurs during one *passage* of the algorithm, where a passage is the time between when a process leaves the NCS and next returns to it. This is a good measure of an ME algorithm's efficiency, since an RMR involves traversing the processor-to-memory interconnect, which can be much slower than accessing a variable locally [38].

An ME algorithm that has bounded RMR complexity is typically called a *local-spin* ME algorithm. The term refers to the behaviour of a process while it is waiting to enter the CS. A process must wait to enter the CS, for example, when another process is already in the CS. While a process waits, it executes a loop in which it checks some set of variables repeatedly until it can advance. If the algorithm has bounded RMR complexity, then the set of variables that the process references in the loop must be locally accessible. In this situation, the process is said to be locally "spinning". The algorithms that we present in this thesis are all local-spin algorithms.

The RMR complexity of an ME algorithm may depend on the number of processes contending for access to the CS. *Point contention* describes this quantity precisely; for our purposes it is defined as the maximum number of processes simultaneously outside of the NCS at any point during a passage. An ME algorithm whose RMR complexity grows gradually with point contention is known as *adaptive* (to point contention). Point contention is often denoted by k, however this conflicts with our use of k in k-exclusion. Instead, we use c to denote this quantity throughout this thesis.

1.6 Synchronization Primitives

In the shared-memory model, processes communicate by using various synchronization primitives to read and update shared variables. The weakest form of primitives are non-atomic reads and writes. Intuitively, if a process tries writing a variable "solo", i.e., without any other processes reading or writing the same variable concurrently, then the value written will be stored correctly in the variable. If another process tries reading (or writing) the same variable concurrently with the write, then the value read (or written) is arbitrary. There also exist stronger synchronization primitives such as atomic reads and writes, in which processes can read or write shared variables atomically. The effect of an atomic read or write on a shared variable appears to be instantaneous. More precisely, atomic reads and writes of variables in this thesis behave the same as reads and writes of atomic registers as defined by Lamport [34]. Non-atomic reads and writes of variables behave similar to reads and writes of safe registers, also as defined by Lamport. Lamport, however, only studied multi-reader single-writer safe registers, i.e., the case where there was no possibility for concurrent writes. In this thesis, our definition of non-atomic reads and writes is more accurately compared to what would be called multi-reader *multi-writer* safe registers, although we are not aware of any formal definition of such registers in the literature. We define the behaviour of non-atomic reads and writes more precisely in Chapter 5.

Note that when using atomic reads and writes in a mutual exclusion algorithm, we effectively assume that mutual exclusion has already been solved at the level of reads and writes. That is, each time a process reads or writes a variable, the read or write takes place within a critical section. This observation leads us to the following intriguing question: Is it possible to solve mutual exclusion without assuming mutual exclusion at the level of reads and writes, i.e., using non-atomic reads and writes? It turns out that it is. We mention a couple of such algorithms in the next section where we discuss previous work. There are also more general techniques known as register constructions

[33, 34], which allow the "construction" of atomic registers from safe registers, i.e., allow atomic reads and writes to be simulated by using only non-atomic reads and writes. These techniques can be used to transform any algorithm that uses only atomic reads and writes to an algorithm that uses only non-atomic reads and writes. The drawback of this approach is that it is inefficient, since it increases the RMR and space complexity of an algorithm by a factor of at least $\Omega(N)$. As such, we do not use these techniques in the design of any algorithms in this thesis.

Beyond atomic reads and writes, there are even stronger primitives like Compare&Swap and Fetch&Add. These primitives allow a process to atomically read and update a variable based on the result of the read. We do not make use of these stronger primitives in any of the algorithms presented in this thesis. Instead, we use only atomic reads and writes, with the exception of the chapter on k-exclusion, where a number of the algorithms that we present use only non-atomic reads and writes.

1.7 Previous Work

The mutual exclusion problem was first formulated and solved by Dijsktra [16]. However, his solution did not satisfy lockout freedom. Instead, his solution satisfied deadlock freedom.

Knuth [31] was the first to present a mutual exclusion algorithm that satisfied lockout freedom.

The FCFS property for mutual exclusion was originally formulated by Lamport in [32], where he also presented the first ME algorithm that satisfied this property. This algorithm is known as the Bakery algorithm, and is also the first ME algorithm to use non-atomic reads and writes. It is not local-spin, and suffers from the drawback that it uses shared variables that can grow without bound. Hadzilacos and Lycklama [36] presented a non-local-spin FCFS ME algorithm that uses only non-atomic reads and

writes, and only requires five shared bits per process, meaning that shared variables are bounded, unlike the Bakery algorithm.

Yang and Anderson (YA) [43] presented the first mutual exclusion algorithm that uses only atomic reads and writes and has RMR complexity $O(\log N)$. Their algorithm does not satisfy the FCFS property, however. Anderson and Kim [3] presented the first mutual exclusion algorithm that uses only non-atomic reads and writes and has RMR complexity $O(\log N)$. This algorithm does not satisfy the FCFS property, either.

Jayanti [25] presented the first FCFS mutual exclusion algorithm with RMR complexity $O(\log N)$, although his algorithm requires the use of LoadLinked and StoreConditional in addition to atomic reads and writes.

Taubenfeld [42] presented the Black-White Bakery algorithm, which is a variant of the Bakery algorithm that is local-spin, adaptive, and uses only shared variables of bounded size. Like the Bakery algorithm, it satisfies FCFS, however it differs in that it requires reads and writes be atomic. Its RMR complexity is $O(c^2)$, where c is point contention.

Scott and Scherer [41] proposed the first local-spin abortable mutual exclusion algorithm, but it suffers from the drawback that a process executing the abort protocol has to potentially wait for other processes before it can make progress. That is, it does not satisfy bounded abort, which defeats the purpose of providing abort functionality. Later Scott [40] corrected this deficiency, but the resulting algorithm has unbounded RMR complexity and requires unbounded space in the worst case. Both of the two preceding algorithms also require the use of strong synchronization primitives.

Jayanti [26] presented the first local-spin FCFS abortable mutual exclusion algorithm that satisfies bounded abort. It has $O(\min(c, \log N))$ RMR complexity, and uses LoadLinked and StoreConditional.

Group mutual exclusion (GME) was first introduced and solved by Joung [28]. His solution is not local-spin. Keane and Moir [29] presented a local-spin GME algorithm, except that their solution does not satisfy the concurrent entering property. The reason

for this is that their solution uses ordinary mutual exclusion as a building block in the trying and exit protocols of the algorithm. If multiple processes requesting the same session attempt to execute the trying protocol concurrently, some processes may have to wait for an unbounded amount of time as other processes execute through the ME building block. Danek and Hadzilacos [14] presented the first local-spin GME algorithm. Their algorithm is in the form of a transformation from FCFS abortable mutual exclusion to GME. The transformation itself uses only atomic reads and writes. Prior to the present work, however, the only known FCFS abortable ME algorithm was that of Jayanti [26], which uses LoadLinked and StoreConditional. Hence, the GME algorithm that is the result of the transformation also requires LoadLinked and StoreConditional.

More recently, Bhatt and Huang [6] presented a group mutual exclusion algorithm that has $O(\min(c, \log N))$ RMR complexity in the CC model. Their algorithm uses LoadLinked and StoreConditional.

The k-exclusion problem was first studied by Fischer et al. [19]. They presented a number of algorithms, none of which is local-spin. The correctness of their algorithms also rely on the use of "atomic actions". Atomic actions are segments of code that Fischer et al. assume execute atomically. The way atomic actions are presented in their algorithm is by surrounding the code segment with the trying and exit protocols of an ordinary ME algorithm. Fischer et al. concede that these code segments cannot actually be executed atomically in practice. As a result, their k-exclusion algorithm does not satisfy starvation freedom: if a single process halts inside an "atomic action", it can prevent all other processes from making progress. To get around this dilemma, they assume that processes do not crash while executing an atomic action.

Afek et al. [1] presented a k-exclusion algorithm inspired by the Bakery algorithm. Unlike the Bakery algorithm, this algorithm uses bounded shared variables and requires the use of atomic reads and writes. Its structure turns out to be similar to several of the k-exclusion algorithms presented in Chapter 5. The main differences between our

algorithms and the algorithm of Afek et al. are that the latter is not local-spin, and we also present variants of our k-exclusion algorithms that make use of only non-atomic reads and writes.

Several k-exclusion algorithms have been presented [39, 22, 15] that do not satisfy starvation freedom as we defined it. Rather, they satisfy weaker progress properties that depend on processes not crashing. In particular, the algorithm of Peterson [39] was not originally presented as a k-exclusion algorithm, but it is an elementary exercise to convert it into one [37]. This algorithm satisfies a restricted form of the deadlock freedom property for k-exclusion in which processes are only allowed to crash in the CS. The algorithm of Gottlieb et al. [22] satisfies a restricted form of starvation freedom in which processes are not allowed to crash anywhere outside the NCS. The algorithm of Danek and Lee [15] is local-spin, however, like the algorithm of Gottlieb et al., starvation freedom depends on there being no process crashes. Also, it has $\Theta(N \log N)$ RMR complexity in the worst case, which we improve upon with the k-exclusion algorithms in this thesis. Excluding these exceptions, our discussion of k-exclusion algorithms in this thesis is restricted to algorithms satisfying all of k-exclusion, starvation freedom, and bounded exit.

Burns and Peterson [8] presented a k-exclusion algorithm that requires atomic reads and writes, but is not local-spin. Dolev et. al [18] presented a k-exclusion algorithm that requires only non-atomic reads and writes, but it is also not local-spin.

Bhatt and Jayanti [7] presented an algorithm that uses a failure detector [10] to solve a variant of k-exclusion. In the variant they study, at most k live processes can be in the CS at the same time. In other words, more than k processes can be in the CS at the same time, as long as the excess number of processes beyond the k live processes have all crashed. The structure of their k-exclusion algorithm has some similarities to the k-exclusion algorithms presented in this thesis. The major difference is that they require the use of a failure detector. Other differences include the fact that their algorithm is not local-spin, and that they do not have a variant that works with non-atomic reads

Reference	RMR Complexity	Instructions Used	Starvation Free?
Fischer et al. [19]	∞	AA	Y
Fischer et al. [20]	∞	AA	Y
Dolev et al. [18]	∞	NARW	Y
Afek et al. [1]	∞	ARW	Y
Peterson (CC) [39]	$\Theta(N^3 - Nk^2)$	ARW	N
Peterson (DSM) [39]	∞	ARW	N
Burns and Peterson [8]	∞	ARW	Y
Gottlieb et al. [22]	∞	ARW, Fetch&Add	N
Bhatt and Jayanti [7]	∞	ARW, FD	Y
Anderson and Moir (CC/DSM) [2]	$\Theta(k \log (N/k))$	ARW, Fetch&Add, Test&Set, Compare&Swap	Y
Anderson and Moir (CC/DSM) [2]	$\Theta(c)$	ARW, Fetch&Add, Test&Set, Compare&Swap	Y
Danek and Lee (CC/DSM) [15]	$\Theta(k \log N)$	ARW	N

Table 1.1: Known k-exclusion algorithms. AA = Atomic Actions, ARW = Atomic Reads & Writes, NARW = Non-atomic Reads & Writes, FD = Failure Detector

and writes.

The only previously known local-spin k-exclusion algorithms satisfying starvation freedom are by Anderson and Moir [2]. Their algorithms have RMR complexity $\Theta(k \log (N/k))$ and $\Theta(c)$, which turns out to be $\Theta(N)$ in the worst-case, for any k that is a constant fraction of N (e.g., for $k = N/2$). Unlike the algorithms in this thesis, Anderson and Moir's algorithms use strong synchronization primitives such as Fetch&Add, Test&Set, and Compare&Swap in addition to atomic reads and writes.

We summarize the preceding discussion on k-exclusion algorithms in Table 1.1.

1.8 Contributions and Thesis Outline

In the next chapter we describe in more detail the asynchronous shared-memory model that we use throughout this thesis, and how we model atomic read and write operations. We also describe several CC model variants, and describe how the correctness properties for the different variants of mutual exclusion introduced in this chapter relate to the computational model. We defer a description of our non-atomic model to Chapter 5, which is the only chapter that uses it.

In Chapter 3, we present the first known algorithm for FCFS mutual exclusion that has RMR complexity $O(\log N)$ in the DSM and CC models, and uses only atomic reads and writes. All previously known algorithms for FCFS mutual exclusion either have super-logarithmic RMR complexity or use synchronization primitives that are stronger than atomic read and write. Our algorithm is also adaptive to point contention. More precisely, the RMR complexity of our algorithm is $\Theta(\min(c, \log N))$, where c is the point contention.

In Chapter 4, we present a novel transformation from abortable mutual exclusion to FCFS abortable mutual exclusion that uses only atomic reads and writes. This transformation, in conjunction with results by Lee [15, 35], yields the first known local-spin

FCFS abortable mutual exclusion algorithm that uses only atomic reads and writes. Furthermore, the transformation, in conjunction with results by Danek and Hadzilacos [14], yields the first known local-spin group mutual exclusion algorithm that uses only atomic reads and writes. Both algorithms that result from the transformation have RMR complexity $O(N)$ in the DSM and CC models, which is tight for the DSM model [12].

Chapter 5 contains the first known local-spin k-exclusion algorithms that use only atomic reads and writes. All algorithms have RMR complexity $O(N)$ in both the DSM and CC models, and satisfy the k-FCFS property. One variant of the algorithms additionally satisfies the FIFE property. We also present a version of the FIFE k-exclusion algorithm that works with non-atomic reads and writes. The high-level structure of all our algorithms is inspired by Lamport's famous Bakery algorithm.

Problem	Upper Bound (RW)	Upper Bound (SS)	Lower Bound (RW)	Lower Bound (SS)
ME	$O(\log N)$ [43]	$O(1)$ [38]	$\Omega(\log N)$ [5]	$\Omega(1)$
FCFS ME	$O(\log N)$ (Ch. 3)	$O(1)$ [38]	$\Omega(\log N)$ [5]	$\Omega(1)$
FCFS Abortable ME	$O(N)$ (Ch. 4)	$O(\log N)$ [26]	$\Omega(\log N)$ [5]	$\Omega(1)$
k-exclusion	$O(N)$ (Ch. 5)	$\Theta(k \log (N/k))$ [2]	$\Omega(\log N)$ [5]	$\Omega(1)$

Table 1.2: Comparison of upper bounds in this thesis to best-known lower-bounds. RW = Reads and Writes; SS = Strong Synchronization Primitives

Table 1.2 summarizes the results in this thesis and provides a comparison with the best known lower-bounds. There is a gap between some of the upper-bounds and lower-bounds, which have yet to be closed. We conclude in Chapter 6 with a discussion of some of these issues.

Chapter 2

Preliminaries

In this chapter we describe the asynchronous shared-memory model of computation with atomic reads and writes that we use in the majority of this thesis. We defer a description of our computation model with non-atomic reads and writes to Chapter 5, which is the only chapter where we use it.

We also describe a number of different cache-coherent model variants, and describe more precisely, in the context of our model, the meaning of the correctness properties for the mutual exclusion variants introduced in the first chapter.

2.1 Asynchronous Shared-Memory Model With Atomic Reads and Writes

Our goal is to model the execution of processes in an asynchronous shared-memory multiprocessor computer *system*. For our purposes, a *system* consists of a set of N processes and a set of shared variables to which processes have access. Each shared variable in the system has an *initial value* that corresponds to the value it is assigned before being accessed by any processes. A *process* is an automaton that has a *private state*, which is reflected by the values of a set of *private variables* (i.e., variables to

which only that process has access). In this thesis, we define processes informally using pseudocode.

The (global) *state* of the system specifies an assignment of values to all the shared variables and all the private variables of each process. The *initial state* of the system consists of the assignment of initial values to shared variables and the private state of each process prior to any process taking any *steps*. A *step* by a process consists of a private computation, along with the read or write of exactly one shared variable.

Informally, an *execution* describes the sequence of steps taken by processes in the system and how the state of the system evolves as a result of these steps. More precisely, an execution (also called an *execution history*) is a sequence (possibly infinite) of alternating steps and states that begins with the initial state s_0, and ends in a state (if finite). In an execution history $H = s_0, t_0, s_1, ...$, we use s_i to represent the i'th state, and t_i to represent the i'th step by some process. We say that H is *valid* if for any contiguous subsequence $..., s_i, t_i, s_{i+1}, ...$ of H, two properties hold: First, if t_i is performed by process p, then p's step is consistent with its program, i.e., t_i is a possible step for p given p's private state in state s_i. Second, the state s_{i+1} is identical to s_i, except that the value of the shared variable and private variables written in t_i is as prescribed by t_i.

We henceforth assume that all executions dealt with in this thesis are valid executions. Also, we assume that processes execute *asynchronously*, meaning that between any two steps of some process in an execution history, there can be an arbitrary number of steps taken by other processes.

A process p *crashes* in an infinite execution H if it stops taking steps while outside the NCS, i.e., there exists a state in H after which p does not take any steps and after p's last step, p is outside of the NCS.

An *algorithm* is a way of specifying the system (i.e., it is a specification of the program for each process in the system, and the initial state of the system). An "execution of an algorithm" or an "execution generated by an algorithm" is shorthand for saying an

execution of the system specified by the algorithm.

2.1.1 Notation

States of the system in an execution history are sequentially numbered starting at 0. The number assigned to a state is referred to as its *index*, and allows us to uniquely identify each state in an execution history. An index is a non-negative integer $i \in \mathbb{N}$. We use $H[i]$ to denote the state with index i. Each step is also uniquely identified by the index associated with the state immediately preceding it. Step i is the step that occurs immediately after the state with index i and before the state with index $i + 1$. Index numbers are not accessible by processes and are only used as a convenient way of referring to parts of execution histories in our proof of correctness.

For any $i, j \in \mathbb{N}$, where $j > i$, we use the notation $H[i, j]$ to denote the execution *subhistory* of H from the state at index i to the state at index j (inclusive).

We use the notation $p@i$ to indicate that process p is in a state such that its next step will correspond to the execution of a step at line i in the pseudocode for the algorithm, and the notation $p@\{i..j\}$ to indicate that $p@k$ for some k in the range $i..j$.

We use the notation $p.v$ to refer to a private variable v that is owned by process p.

2.2 Correctness Properties and Proofs

In Chapter 1 we introduced a number of variants of mutual exclusion and their correctness properties. These correctness properties were stated informally without reference to the model of computation being used. We explain here what these properties mean in the context of our model.

Let $\mathbb{A}$ be an algorithm for a system with N processes. A correctness property is a

predicate of an execution history H. Let $\mathbb{H}$ be the set of all execution histories generated by $\mathbb{A}$. We say that $\mathbb{A}$ *satisfies* a correctness property if $\forall H \in \mathbb{H}$, the correctness property is true for H.

For example, the correctness properties for ordinary mutual exclusion are as follows:

Mutual Exclusion: $\mathbb{ME}(H)$: For all indices i, at most one process is in the CS in state $H[i]$.

Lockout Freedom: $\mathbb{LF}(H)$: If H is infinite and no processes crash in H, then for all indices i, and all processes p, if p is in the TP in state $H[i]$, then there exists an index $j > i$ such that p is in the CS in state $H[j]$.

$f(N)$**-Bounded Exit:** $\mathbb{BE}(H, f)$: For all indices i, and all processes p, if p is in the EP in state $H[i]$, then if p takes at least $f(N)$ steps after state $H[i]$, then there exists an index $j > i$ such that p is in the NCS in state $H[j]$ and p takes at most $f(N)$ steps in $H[i, j]$.

Let $\mathbb{F}$ be the set of functions dependent only on N. We say that the N-process algorithm $\mathbb{A}$ satisfies mutual exclusion (resp. lockout freedom, bounded exit) if $\exists f \in \mathbb{F}$, $\forall H \in \mathbb{H}$ such that $\mathbb{ME}(H)$ (resp. $\mathbb{LF}(H)$, $\mathbb{BE}(H, f)$) is true. We say that $\mathbb{A}$ is a correct mutual exclusion algorithm if it satisfies mutual exclusion, lockout freedom, and bounded exit.

$\mathbb{A}$ may sometimes satisfy a correctness property only *under certain conditions*[1], where a condition is a predicate of an execution history H. We say that $\mathbb{A}$ satisfies a correctness property under a given condition, if $\forall H \in \mathbb{H}$ if the given condition is true for H, then the correctness property is also true for H.

When proving an algorithm correct, we generally use the less formal statements of its correctness properties, such as those in Chapter 1, with the understanding that they

[1]This terminology allows several correctness properties in Chapter 3 to be stated more naturally.

have more precise counterparts of the style described above. In particular, we avoid explicitly referencing execution histories in our proofs unless necessary for the clarity of the argument. The intent behind this is to avoid overly complex notation, and as a result, impart a higher level of intuition as to why our algorithms are correct.

Also, when describing steps executed by a process in a proof, we avoid mentioning the specific passage the process is executing. Processes can execute multiple passages of an algorithm, and so to be strictly accurate, we should also specify the passage it is executing. However, to avoid overly complex notation, if we do not have to reason across multiple passages, we omit specifying the passage and assume that it is the "current" (i.e., latest) one with respect to some state in the execution.

2.3 Variants of the CC Model

As explained in the introduction, there are two ways of defining how shared memory works: the DSM model, and the CC model. We explain here the CC model variants relevant to this thesis.

The following features are common to all variants of the CC model: (i) there is a "main memory", which a process can access by making a remote memory reference; and (ii) each process has a local cache of unbounded size, which a process can access by making a local memory reference. The different variants of CC models can be classified along two different dimensions: the protocol used for handling updates to main memory (*write-through* or *write-back*), and the protocol used for enforcing cache consistency (*write-invalidate* or *write-update*).

With a write-through/write-invalidate protocol, any time a process writes a shared variable, it *writes it through* to main memory, hence making a remote memory reference. Whenever a process reads a shared variable, it first attempts to read a copy of the variable from its cache. If a copy does not exist in the process's cache, the process will read the

variable from main memory, thereby incurring a remote memory reference, and then will cache a copy of the variable locally. A cached copy of the variable v remains in a process p's cache until the copy is *invalidated*, at which time it is effectively erased from p's cache. All cached copies of v are invalidated whenever any process writes v to main memory. Note that when a process writes a variable, it incurs only one remote memory reference, even if the write results in multiple cached copies of the variable being invalidated.

When a write-back protocol is used with a write-invalidate protocol, writes are initially cached and the cached copies are written back to main memory at some later point. The criterion used for deciding when to write back a cached copy of a variable to main memory has to do with a *mode* flag associated with the copy. A variable can be cached in *exclusive mode* or *shared mode*. If a cached copy of a variable is in a process p's cache in exclusive mode, then any writes by p are made to its cache. However, if a cached copy of variable v is not in a process p's cache, or it is there in shared mode, then a write by p does the following: it first invalidates all other copies of v, then it writes to main memory any copy of v cached in exclusive mode by another process, and finally it creates a copy of v in its own cache in exclusive mode. Reads are also slightly more complex: if a process p reads a variable v that is not in its cache, then p writes to main memory any copy of v cached in exclusive mode by another process q, changes the mode of the copy of v in q's cache to shared, and then copies v from main memory into its own cache. If p reads a variable v that is already in its cache in either mode, then p reads the cached copy locally without changing the copy's mode flag. Like the write-through/write-invalidate model, a cached copy of a variable remains in a process's cache until it is invalidated.

In the write-update protocol, whenever a process writes a variable v, it updates the contents of any other caches that currently store v. Write-update protocols are complex and harder to implement in practice, and thus are not common. The main difficulty with this protocol is that it must broadcast writes to every cache, and as a result, takes up more bandwidth [23]. We do not consider this variant of the CC model further in this

thesis.

For the rest of this thesis, when dealing with the CC model we assume that a write-through/write-invalidate protocol is used. Our RMR complexity results apply equally to the CC models that use write-back/write-invalidate protocols. This is because the number of RMRs incurred using write-through/write-invalidate is an upper-bound for the number of RMRs incurred using the write-back/write-invalidate model.

Chapter 3

Efficient FCFS Mutual Exclusion

In this chapter we present the first known algorithm for FCFS mutual exclusion that uses only atomic reads and writes and has $O(\log N)$ RMR complexity. Our algorithm is also adaptive to point contention. More precisely, our algorithm has $\Theta(\min(c, \log N))$ RMR complexity, where c is the point contention.

Note that the properties satisfied by the algorithm that we present in this chapter cannot be duplicated by combining the FCFS abortable mutual exclusion algorithm of Jayanti [26], which uses LoadLinked and StoreConditional and has $O(\min(c, \log N))$ RMR complexity, with the results of Golab et al. [21]. Golab et al. have presented an implementation of a wide class of primitives, which includes LoadLinked and StoreConditional, that uses only reads and writes and requires only a constant number of RMRs for each operation of the implemented primitive. Using this implementation they show how to transform algorithms that rely on this class of primitives into algorithms that use only reads and writes, with only a constant-factor increase in the number of RMRs.

Prima facie this result appears to immediately imply the main result of this chapter: Applying the transformation of Golab et al. to Jayanti's algorithm would appear to yield a $O(\min(c, \log N))$ RMR complexity adaptive, FCFS mutual exclusion algorithm that

uses only reads and writes. This is not so, however: Although the transformation of Golab et al. preserves properties such as mutual exclusion and lockout freedom, it does not necessarily preserve properties such as FCFS or bounded exit. This is because the transformation introduces potentially unbounded waiting (albeit on locally accessible variables, so that RMR complexity is preserved within a constant factor), and may therefore fail to preserve properties that require bounded termination, as FCFS and bounded exit do. In fact, applying the Golab et al. transformation to Jayanti's algorithm results in an algorithm that satisfies mutual exclusion and lockout freedom, but not FCFS or bounded exit.

3.1 FCFS Algorithm and High-level Description

The new FCFS mutual exclusion algorithm is given in Figure 3.1 and has the following high-level structure. In the doorway, a process receives a ticket from a ticket dispenser (line 4). The dispenser is similar to a modular atomic counter, which returns tickets with increasing values from a bounded interval. As the dispenser is not actually atomic, processes that invoke the dispenser concurrently may receive the same ticket. Also, even though the dispenser returns tickets from a bounded interval, meaning that the dispenser eventually "wraps around", the interval is large enough to ensure that tickets are not reused too soon. After a process p obtains a ticket, it enters the waiting room (lines 5–16) where it adds itself to a priority queue (Q) ordered by ticket (line 11). To ensure that FCFS is not violated, p waits to reach the front of the queue before entering the CS (line 15). Once p is done with the CS, p removes itself from the queue (line 18), and notifies its successor (lines 20–22).

The priority queue has standard operations Insert and FindMin, and its entries are pairs of the form (process ID, ticket). We also define the operation Remove, which allows any element to be removed from the priority queue. Insert is idempotent, and Remove

Figure 3.1: FCFS mutual exclusion algorithm for process $p \in \{1, ..., N\}$.

shared variables:
Set: *SpecialSet*
Q: *PriorityQueue*
Head: **array**[1..*N*] of **Boolean**

(In the DSM model, *Head*[*p*] is local to process *p*.)

private variables:
ticket: **integer** from the set $\{0, .., 7N - 1\}$
tmp_id: **integer**

```
1  loop
2      NCS
3      Set.InsertSelf()                                  // Doorway begins.
4      ticket := ObtainTicket()                            // Doorway ends.
5      LOCK()
6          Head[p] := false
7          tmp_id := Set.RemoveSelf()
8          if tmp_id ≠ ⊥ then
               // Enqueue process tmp_id with ''dummy'' ticket.
9              Q.Insert(tmp_id, −1)
10         Q.Remove(p, −1)            // Remove (p, −1) from queue if present.
11         Q.Insert(p, ticket)          // Reinsert p with ''proper'' ticket.
12         tmp_id := Q.FindMin()        // Get the head process in the queue.
13         Head[tmp_id] := true            // Notify head process to advance.
14     UNLOCK()
15     await Head[p] = true           // Wait to reach the head of the queue.
16     LOCK()
17         CS                                          // The critical section.
18         Q.Remove(p, ticket)          // Remove p from the priority queue.
19         DoneWithTicket()
20         tmp_id := Q.FindMin()
21         if tmp_id ≠ ⊥ then
22             Head[tmp_id] := true         // Notify next process to advance.
23     UNLOCK()
24 end loop
```

has no effect if attempting to remove an item that is not in the queue. FindMin() returns the process ID whose corresponding ticket is minimal (i.e., the head element), or $\perp$ if the queue is empty. An *auxiliary lock* (lines 5, 14, 16, 23) is used to serialize operations on the priority queue, which allows us to implement the priority queue with logarithmic step complexity using elementary data structure techniques.

It turns out that to protect the priority queue in the exit protocol it is not strictly necessary for the auxiliary lock to be acquired at line 16 (before the CS). The algorithm could be modified to acquire the lock immediately after the CS and it would still be correct. In this case, however, the bounded exit property would not be satisfied. As bounded exit can be a desirable property, the algorithm is designed to satisfy it.

Processes use the Boolean array *Head* to notify another process when it becomes the head of the queue. A process can become the head of the queue after another process removes itself from the queue in the exit protocol (line 18), or after the queue is modified in the waiting room (lines 10–11).

The algorithm contains additional features, not described above, to handle the following race condition: process p finishes the doorway before q starts the doorway, but then q adds itself to Q before p. By the FCFS property, p should enter the CS before q, but until p is added to Q, q cannot tell (from the state of Q alone) whether it should enter the CS before or after p. To prevent q from entering the CS prematurely, special "dummy tickets" and a shared object, *Set*, of a set-like type called *SpecialSet* are used. At the beginning of the doorway, at line 3, a process q adds itself to *Set*. In the waiting room, at line 7, q removes itself from *Set*, and also learns the ID of one other process $p \neq q$ in *Set*, if it exists ($\perp$ otherwise). If p exists, then p must be in the trying protocol at or before the lock at line 5; p cannot be past this point, as q holds the auxiliary lock while q is at lines 6–13. Process q adds p to Q at line 9 with a "dummy" ticket -1, which is smaller than any "proper" ticket returned by the ticket dispenser at line 4. The insertion of p into Q in this way guarantees that p will be ahead of q in Q until p executes

the locked code at lines 6–13, where it replaces its dummy ticket in Q with its proper ticket (lines 10–11). This ensures that q cannot advance into the CS prematurely.

The $\Theta(\min(c, \log N))$ RMR complexity of the algorithm arises as follows. The only lines where a super-constant number of RMRs can be made are at the lines where a process accesses the *SpecialSet* object, the ticket dispenser, the priority queue Q, or the auxiliary lock. For the auxiliary lock we use Kim and Anderson's non-FCFS mutual exclusion algorithm [30], which has RMR complexity $\Theta(\min(c, \log N))$. We implement the priority queue using elementary data structure techniques, resulting in an implementation with $O(\log c)$ step complexity (and hence RMR complexity). Lastly, for the *SpecialSet* and ticket dispenser, we use implementations that have $O(\min(c, \log N))$ step complexity (and hence RMR complexity) [13]. Thus, the overall RMR complexity of the algorithm presented here is $\Theta(\min(c, \log N))$.

A more precise description of the properties that the *SpecialSet* and ticket dispenser building blocks must satisfy is given in the following sections.

3.2 *SpecialSet* Specification

In this section, we describe the data structure *Set* used in our mutual exclusion algorithm. We refer to this data structure as a *SpecialSet*, because it only supports certain specialized set operations.

A *SpecialSet* stores a set of process IDs, which is initially empty. Processes manipulate the set only by adding and removing their own IDs through the operations InsertSelf() and RemoveSelf(), which must be accessed according to the following condition:

Condition 1. *The calls to* InsertSelf() *and* RemoveSelf() *made by any process occur in an alternating sequence, beginning with* InsertSelf().

We refer to the time between the start of a call to InsertSelf() and the next time RemoveSelf() finishes as a *Set-passage*.

Function InsertSelf() returns nothing to the caller, whereas RemoveSelf() returns information about set elements other than the caller's own ID. In order to specify the response of RemoveSelf() more precisely, we first establish some definitions:

Definition 1. *A process is* in the set *if and only if it has completed a call to* InsertSelf(), *but has not subsequently completed a call to* RemoveSelf().

Definition 2. *A process is* participating in the set *if and only if it is either in the set or it is executing* InsertSelf().

The value returned by RemoveSelf() is more precisely specified as follows:

Specification 1. *If a process p calls* RemoveSelf() *and this call terminates, then the value returned by* RemoveSelf() *is either $\perp$ or some process ID q, according to the following rules:*

- *If* RemoveSelf() *returns $q \neq \perp$, then $q \neq p$ and q was participating in the set at some point during p's call to* RemoveSelf().
- *If* RemoveSelf() *returns $\perp$, then no process other than p was continuously in the set throughout p's call to* RemoveSelf().

Danek and Golab [13] presented an algorithm for *SpecialSet* that has logarithmic step complexity (and hence RMR complexity) and is also adaptive. More precisely, this implementation satisfies the following property:

Specification 2. *For any process p, and any Set-passage by p, p takes at most $O(\min(c', \log N))$ steps in* InsertSelf() *and* RemoveSelf(), *where c' denotes the maximum number of processes executing a Set-passage simultaneously at any point during p's Set-passage.*

We use this implementation in the FCFS mutual exclusion algorithm in Figure 3.1, where the following restriction on concurrency applies:

Condition 2. *Operation* RemoveSelf() *is executed in mutual exclusion.*

The *SpecialSet* implementation in [13] takes advantage of this condition. (We are not aware of any implementation satisfying the given specification that does not require this condition.)

The following result appears as Theorem 6.12 in [13]:

Theorem 3.1. *The SpecialSet algorithm presented in [13] satisfies Specification 1 under Conditions 1 and 2.*

The following theorem appears as Theorem 6.13 in [13]:

Theorem 3.2. *The SpecialSet algorithm presented in [13] satisfies Specification 2 under Conditions 1 and 2.*

3.3 Ticket Dispenser Specification

Our mutual exclusion algorithm uses numbered tickets, like Lamport's bakery algorithm [32]. Tickets are obtained by calling function ObtainTicket(), which is used in conjunction with function DoneWithTicket() according to the following condition:

Condition 3. *The calls to* ObtainTicket() *and* DoneWithTicket() *made by any process occur in an alternating sequence, beginning with* ObtainTicket().

We refer to the time between the start of a call to ObtainTicket() and the next time DoneWithTicket() finishes as a *dispenser-passage*.

We can think of ObtainTicket() as returning a (not necessarily unique) element of some set of tickets, and DoneWithTicket() as cleaning up some internal state once a process is done using a particular ticket. (Using a pair of functions in this way makes the ticket dispenser somewhat more complex to specify, but easier to implement.)

Definition 3. *We say that a process is* participating in the ticket dispenser *if and only if it has started executing* ObtainTicket(), *but has not completed its subsequent call to* DoneWithTicket().

Informally, we say that a process *holds* a ticket x if it is participating and its last call to ObtainTicket() ran to completion with response x. We say that a ticket is *active* if and only if it is held by some process, otherwise it is *inactive*.

Tickets generated by the dispenser satisfy the following basic properties:

Specification 3. *(a) The domain of tickets is the set of integers modulo mN for some integer $m \geq 2$.*

(b) At any time, active tickets are confined to some interval of at most $mN/2$ integers that are consecutive modulo mN.

We will use 3 (a) and (b) as follows to define a total order on tickets that are simultaneously active. Given two active tickets i and j, where $i < j$, we will say that i is *less than* j (denoted $i \lhd j$) if $j - i < mN/2$, otherwise we will say that i is *greater than* j (denoted $i \rhd j$). Finally, if $i = j$ then we will say i is *equal to* j. (We will also use $\trianglelefteq$ and $\trianglerighteq$ to denote weak inequalities.) For technical reasons, we also define a special *dummy* ticket, denoted -1, which is less than any active ticket. We say that two tickets are *comparable* either if they are simultaneously active, or if one or both is -1. Otherwise we say two tickets are *incomparable*. Finally, note that our mutual exclusion algorithm compares tickets only implicitly, inside the priority queue.

Having defined an ordering among simultaneously active tickets, we are now ready to specify the remaining correctness properties of the ticket dispenser.

Specification 4. *Whenever distinct processes p and q hold tickets t_p and t_q simultaneously, and these tickets were generated by calls C_p and C_q (respectively) to* ObtainTicket(), *then tickets t_p and t_q are related as follows: if C_p completed before C_q, then $t_p \lhd t_q$.*

Danek and Golab [13] presented a ticket dispenser algorithm that has logarithmic step complexity (and hence RMR complexity) and is also adaptive. More precisely, this implementation satisfies the following specification:

Specification 5. *For any process p, and any dispenser-passage by p, p takes at most $O(\min(c', \log N))$ steps in* ObtainTicket() *and* DoneWithTicket()*, where c' denotes the maximum number of processes executing a dispenser-passage simultaneously at any point during p's dispenser-passage.*

We use this implementation in the FCFS mutual exclusion algorithm in Figure 3.1, where the following restriction on concurrency applies:

Condition 4.

(a) Function DoneWithTicket() *is executed in mutual exclusion.*

(b) If processes p and q are participating simultaneously and hold tickets t_p and t_q, respectively, where $t_p \lhd t_q$, then q does not subsequently call DoneWithTicket() *before p. (In other words, q does not stop participating before p does.)*

(c) If processes p and q are simultaneously inside ObtainTicket()*, then q does not subsequently call* DoneWithTicket() *before p completes* ObtainTicket()*.*

Danek and Golab take advantage of this condition to simplify the ticket dispenser implementation in [13]. (We are not aware of any implementation satisfying the given specification that does not require these conditions.)

We can easily verify by inspection of Figure 3.1 that our ME algorithm uses the ticket dispenser according to Condition 3, and Condition 4 (a). However, it is not clear that our ME algorithm also uses the dispenser according to the latter two parts of Condition 4. Establishing that these conditions are true requires careful reasoning about the ME

algorithm, the details of which are provided in the following section, where we prove the ME algorithm correct.

The following result appears as Theorem 7.12 in [13]:

Theorem 3.3. *The ticket dispenser algorithm presented in [13] satisfies Specifications 3 and 4 under Conditions 3 and 4.*

The following result appears as Theorem 7.13 in [13]:

Theorem 3.4. *The ticket dispenser algorithm presented in [13] satisfies Specification 5 under Conditions 3 and 4.*

3.4 Correctness of FCFS ME Algorithm

In this section, we prove the correctness and analyze the RMR complexity of our FCFS mutual exclusion algorithm, which was presented earlier in Figure 3.1.

Our proof of correctness is broken down into two main sections. The first section provides a general proof of correctness in which the only assumption that we make about the *SpecialSet* and ticket dispenser used in the algorithm is that they satisfy the specifications outlined in Sections 3.2 and 3.3, respectively. In particular, at this point, we do not use the implementations of the *SpecialSet* and the ticket dispenser presented in [13], which are correct only under certain conditions. This allows us to reason about the correctness of our algorithm without reference to Conditions 1, 2, 3, and 4.

In the second section, we show that the FCFS ME algorithm uses the *SpecialSet* and ticket dispenser procedures according to Conditions 1, 2, 3, and 4. When used under these conditions, the *SpecialSet* and ticket dispenser implementations presented in [13] are correct, as guaranteed by Theorems 3.1 and 3.3, respectively.

Combining the results of these two sections establishes that the FCFS mutual exclusion algorithm in Figure 3.1 is correct when it uses the *SpecialSet* and ticket dispenser implementations presented in [13].

The first section of our proof is further broken down into three parts. In the first part, we prove that the algorithm satisfies FCFS (Lemma 3.6). This is done by showing that if a process p finishes ObtainTicket() before another process q begins ObtainTicket(), then q does not enter the CS before p. Hence, if p finishes the doorway before q begins the doorway, q does not enter the CS before p.

In the second part, we establish that the algorithm satisfies lockout freedom (Lemma 3.8). This is done by first proving that deadlock freedom holds (Lemma 3.7), and then using the fact that the algorithm satisfies FCFS to conclude that lockout freedom also holds.

In the third part, we prove that our algorithm satisfies mutual exclusion and bounded exit. After establishing the correctness of our algorithm, we analyze its RMR complexity in the last section.

3.4.1 General Proof of Correctness

In this section, we assume an arbitrary correct implementation of *SpecialSet* and the ticket dispenser. That is, we assume that the *SpecialSet* operations, InsertSelf() and RemoveSelf(), satisfy Specifications 1 and 2. We also assume that the ticket dispenser operations, ObtainTicket() and DoneWithTicket(), satisfy Specifications 3, 4, and 5.

3.4.1.1 Part 1 – FCFS

The following lemma is used to prove that the algorithm in Figure 3.1 satisfies FCFS.

Lemma 3.5. *(Smaller-Ticket-First-Served (STFS)) If some processes p and q each execute passages of the algorithm in Figure 3.1 in which they simultaneously hold tickets t_p and t_q, respectively, before entering the CS, and $t_p \lhd t_q$, then q does not enter the CS before p.*

Proof. Suppose, by way of contradiction, that the lemma does not hold. Then there exists passages of p and q in which p and q simultaneously hold tickets t_p and t_q, respectively, before entering the CS, $t_p \lhd t_q$, and q enters the CS before p.

Process q sets $Head[q] = \mathsf{false}$ at line 6, and q cannot set $Head[q] = \mathsf{false}$ again until after executing through the CS and starting a new passage. By assumption, q enters the CS before p, and so q must read $Head[q] = \mathsf{true}$ at line 15 before p enters the CS. The preceding facts mean that there exists some process r (possibly q itself) that sets $Head[q] = \mathsf{true}$: (i) *after* q sets $Head[q] = \mathsf{false}$ at line 6, (ii) before q enters the CS, and therefore (iii) *before* p enters the CS. There are only two places in the algorithm where r can set $Head[q] = \mathsf{true}$: either at line 13 or line 22. These lines are part of the locked segment of code consisting of lines 6–13 and lines 17–22. By this, the fact that line 6, where q sets $Head[q] = \mathsf{false}$, is also part of the locked segment of code, and the fact that r sets $Head[q] = \mathsf{true}$ after q sets $Head[q] = \mathsf{false}$, it follows that r does not set $Head[q] = \mathsf{true}$ until after q finishes executing through lines 6–11. (Recall that r may be q itself. If $r \neq q$, r does not set $Head[q] = \mathsf{true}$ until after q finishes executing through lines 6–13, and hence lines 6–11.) Furthermore, whichever line that r executes to set $Head[q] = \mathsf{true}$ (line 13 or line 22), r's preceding call to Q.FindMin() (line 12 or line 20, respectively), must return q. Lines 12 and 20 are protected by the same lock as lines 13 and 22. Hence, since r sets $Head[q] = \mathsf{true}$ after q executes through lines 6–11, it must also be the case that r's call to Q.FindMin() occurs after q finishes executing through lines 6–11.

Process q removes its dummy ticket and adds itself to the priority queue with its proper ticket (i.e., t_q) at lines 10–11. This and the argument in the preceding paragraph imply that when r calls Q.FindMin(), process q will be in the priority queue with its proper ticket. We now prove that when r calls Q.FindMin(), process p is also in the priority queue with its proper ticket.

Claim 3.5.1. *When r calls Q.FindMin(), process p is in the priority queue with its*

proper ticket (i.e., t_p).

Proof. Suppose, by way of contradiction, that the claim is false. This means that when r calls Q.FindMin(), either p is in the priority queue with a dummy ticket, or it is not in the priority queue at all. The former case is not possible: If p were in the priority queue with a dummy ticket, r's call to Q.FindMin() would have to return a process with a dummy ticket, contradicting that it actually returns q. Thus it must be the case that p is not in the priority queue at all.

Let Γ be the execution subhistory from the step at which q finishes the doorway to when r starts its call to Q.FindMin().

By assumption, process p and q simultaneously hold their tickets t_p and t_q before entering the CS, and $t_p \lhd t_q$. This and Specification 4 (with the roles of p and q interchanged, stated in the contrapositive) imply that p must have started ObtainTicket() at line 4 by the time q finishes ObtainTicket(), and hence by the the beginning of Γ. Furthermore, by assumption, q enters the CS before p, and so p cannot go past line 16 before the end of Γ. Thus, p must be somewhere at lines 4–16 during Γ.

Process p cannot have executed the locked segment of code at lines 6–13 before the end of Γ, otherwise p will be in the priority queue with its proper ticket when r calls Q.FindMin(), contradicting that p is not in the priority queue. Hence, p must be at lines 4–5, and hence also in *Set* (see Definition 1 for what it means for a process to be in *Set*), during Γ.

Let u be the last process to call *Set*.RemoveSelf() at line 7 in Γ. We know that u exists because q must execute *Set*.RemoveSelf() in Γ. To see why this is the case, we observe that: (i) q finishes the doorway at the start of Γ, and (ii) q executes lines 6–11, and hence *Set*.RemoveSelf() at line 7, before r calls Q.FindMin() at the end of Γ.

Process u's call to *Set*.RemoveSelf() will return a value different from $\perp$ by Specification 1 since p is in *Set*, and so u will add a process with a dummy ticket to the priority queue at line 9. Since process u is the last process to execute *Set*.RemoveSelf() in Γ,

it is also the last process to execute through lines 6–11 in Γ. This, and the fact that r cannot call Q.FindMin() until after u has finished lines 6–11, imply that a dummy ticket will be in the priority queue when r calls Q.FindMin(). This contradicts the fact that r's call to Q.FindMin() returns q. □

The set of tickets that are in the priority queue when r calls Q.FindMin() are all active, and so they are totally ordered. (The discussion after Specification 3 describes how a total order can be defined on tickets that are simultaneously active.) Thus when Q.FindMin() returns q to r, we are guaranteed that for any ticket t in the priority queue, $t_q \trianglelefteq t$. By Claim 3.5.1, p is in the priority queue with its proper ticket when r calls Q.FindMin(), and so $t_q \trianglelefteq t_p$. However, this contradicts the assumption that $t_p \triangleleft t_q$. □

Lemma 3.6. *The algorithm in Figure 3.1 satisfies FCFS.*

Proof. Assume that a process p finishes the doorway before a process q starts it, and suppose by way of contradiction that q enters the CS before p. Let t_p and t_q be the tickets that p and q receive from ObtainTicket(), respectively. Since p finishes the doorway before q, and q enters the CS before p, there is a point in the execution in which p and q simultaneously hold t_p and t_q before entering the CS. Moreover, the fact that p finishes the doorway before q starts the doorway means that p finishes ObtainTicket() before q starts ObtainTicket(). Hence by Specification 4, $t_p \triangleleft t_q$. These preceding facts, and Lemma 3.5 (STFS), imply that q does not enter the CS before p. This contradicts the assumption that q enters the CS before p. □

3.4.1.2 Part 2 – Lockout Freedom

To prove that lockout freedom holds, we first prove that deadlock freedom holds. Once we have established that the algorithm satisfies deadlock freedom, we use this in conjunction with the fact that the algorithm satisfies FCFS (Lemma 3.6) to show that lockout freedom also holds.

Lemma 3.7. *The algorithm in Figure 3.1 satisfies deadlock freedom.*

Proof. Suppose, by way of contradiction, that deadlock freedom does not hold. Thus there exists an infinite history H, generated by the algorithm in Figure 3.1, in which no processes crash, such that some process p takes an infinite number of steps in the trying protocol, and after some state in H no process executes through the CS. This, the fact that p does not crash, and the observation that there are no multi-line loops in the trying protocol, imply that there is some line in the trying protocol that p eventually executes repeatedly without advancing further in the algorithm. The *SpecialSet* and ticket dispenser procedures have bounded step complexity by Specification 2 and Specification 5. Consequently, by inspection, the only possible lines in the algorithm that p may not advance past are lines: 5, 14, 15, 16, and 23. By assumption, no process remains in the CS forever, and the auxiliary ME algorithm used at lines 5, 14, 16, 23 satisfies lockout freedom and bounded exit. This implies that no process ever gets stuck in the auxiliary ME algorithm. This means that line 15 is the only line past which p may not advance.

Let q be the last process in H to execute Q.FindMin() (line 12 or line 20). Process q exists, since (i) some process reaches line 15 without advancing past it, and hence must execute line 12, and (ii) there is a state after which no process executes through the CS.

Claim 3.7.1. *When q calls Q.FindMin() for the last time, there are no processes with dummy tickets in the priority queue.*

Proof. Suppose, by way of contradiction, that the claim is false. That is, there exists a process r in the priority queue, Q, with a dummy ticket when q calls Q.FindMin(). The priority queue does not initially contain any elements, and so there must be a step in the execution when some process adds r to Q with a dummy ticket. Consider the last step prior to q's call to Q.FindMin() where this happens, and assume it is done by a process u. By inspection of the algorithm, we see that the only place where u can add r to Q is at line 9. For this line to be executed, u must have received r as a return value from its

preceding call to *Set*.RemoveSelf() at line 7. By Specification 1, we have the following two facts: (i) when u executes line 7, r is in *Set* or is executing *Set*.InsertSelf(), and (ii) $r \neq u$. Furthermore, process r adds itself to *Set* when it calls *Set*.InsertSelf() at line 3 and removes itself when it calls *Set*.RemoveSelf() at line 7. These preceding facts, plus the observation that line 7 is part of a locked segment of code consisting of lines 6–13, imply that $r@\{3..5\}$ holds when u calls *Set*.RemoveSelf().

Process u is the last process prior to q's call to Q.FindMin() to add r to Q with a dummy ticket, and r will remove itself from the queue if it executes through the locked segment of code at lines 6–13. Furthermore, r is in Q with a dummy ticket when q calls Q.FindMin(), and so $r@\{3..5\}$ must be invariant between the start of u's call to *Set*.RemoveSelf() to the end of q's call to Q.FindMin(). After this, r eventually executes through lines 6–13 and calls Q.FindMin() at line 12. This follows from the lockout freedom property of the auxiliary ME algorithm, the fact that all statements in lines 6–13 terminate after a bounded number of steps, and the fact that r does not crash. Thus r calls Q.FindMin() after the last call to Q.FindMin(), which is a contradiction.

□

Claim 3.7.2. *Process q's last call to Q.*FindMin() *does not return $\perp$.*

Proof. Suppose, by way of contradiction, that Q.FindMin() returns $\perp$. This means that Q is empty.

Process q's last call to Q.FindMin() is either at line 12 or at line 20. We consider these two cases separately:

Case 1: q's last call to Q.FindMin() is at line 12.

Immediately before executing line 12, process q executes line 11, where it inserts itself into the priority queue. Since no other process can remove q from Q, and q does not remove itself until q executes line 18, this means Q is non-empty when q executes line 12, contradicting that Q is empty.

Case 2: q's last call to Q.FindMin() is at line 20.

By assumption, some process p reaches line 15 but does not advance past it. Either $p@\{14..15\}$ when q calls Q.FindMin(), or not. In the former case, p will have inserted itself into the priority queue when it last executed line 11. Moreover, p cannot be removed from Q until p executes line 18, and hence Q is non-empty when q executes Q.FindMin(), contradicting that Q is empty. In the latter case, in which $p@\{14..15\}$ is not true when q calls Q.FindMin(), to reach line 15 p will have to execute through lines 6–13 after q has executed line 23, and therefore after q has executed line 20. This means that p will execute Q.FindMin() at line 12 after the last call to Q.FindMin(), which is a contradiction.

□

Claim 3.7.3. *Process q's last call to Q.FindMin() does not return q.*

Proof. Suppose, by way of contradiction, that Q.FindMin() returns q.

Process q's last call to Q.FindMin() is either at line 12 or line 20. We consider these two cases separately:

Case 1: q's last call to Q.FindMin() is at line 12.

In this case, q sets $Head[q] = \mathsf{true}$ at line 13. Hence, q will advance past line 15. By the lockout freedom property of the auxiliary ME algorithm, the fact that the CS is finite, the fact that q does not crash, and the fact that the statements in lines 18–19 terminate after a bounded number of steps, it follows that q will eventually execute line 20. That is, q will eventually execute Q.FindMin() after the last call to Q.FindMin(), which is a contradiction.

Case 2: q's last call to Q.FindMin() is at line 20.

Process q removes itself from the priority queue at line 18, and so the call to Q.FindMin() at line 20 cannot return q, contradicting the assumption that it does.

□

Claim 3.7.4. *If process q's last call to Q.FindMin() returns s, $s \neq \perp$, and $s \neq q$, then process s makes a call to Q.FindMin() after q's last call.*

Proof. Assume process q's call to Q.FindMin() returns s, $s \neq \perp$, and $s \neq q$. Since $s \neq \perp$, the process s is in the priority queue. By Claim 3.7.1, s cannot be in the priority queue with a dummy ticket. This means that when q calls Q.FindMin(), s must have inserted itself with its proper ticket at line 11 and not yet removed itself at line 18, i.e., $s@\{12..18\}$. By the assumption that $s \neq q$, this means s cannot be in any locked segments concurrently with q, and so when q calls Q.FindMin(), $s@\{14..16\}$. After q calls Q.FindMin(), q sets $Head[s] = \mathsf{true}$ (either at line 13 or line 22). No process can set $Head[s] = \mathsf{false}$ after this until s executes its next passage (if any). It follows from this, lockout freedom of the auxiliary ME algorithm, the fact that processes do not crash, and the fact that the statements in lines 18–19 terminate after a bounded number of steps, that after q's last call to Q.FindMin(), process s eventually executes through the CS and reaches line 20 where it calls Q.FindMin(). That is, s calls Q.FindMin() after q's last call to Q.FindMin().

□

By Claims 3.7.2, 3.7.3, and 3.7.4, it follows that some process executes Q.FindMin() after the last call to Q.FindMin(), which is a contradiction. □

Lemma 3.8. *The algorithm in Figure 3.1 satisfies lockout freedom.*

Proof. The lemma follows immediately from the fact that the algorithm satisfies FCFS (Lemma 3.6) and deadlock freedom (Lemma 3.7). □

3.4.1.3 Part 3 – Mutual Exclusion and Bounded Exit

Lemma 3.9. *The algorithm in Figure 3.1 satisfies mutual exclusion.*

Proof. The lemma follows immediately from the fact that the CS is surrounded by the trying and exit protocol of the auxiliary mutual exclusion algorithm (line 16 and line 23). □

Lemma 3.10. *The algorithm in Figure 3.1 satisfies bounded exit.*

Proof. The exit protocol of the auxiliary mutual exclusion algorithm at line 23 satisfies bounded exit, and by Specification 5, the DoneWithTicket() operation at line 19 terminates after a bounded number of steps. Moreover, the priority queue is implemented using elementary (sequential) data structure techniques, and so the priority queue operations at line 18 and line 20 also terminate after a bounded number of steps. By inspection, a process finishes the other lines of the exit protocol in a bounded number of its own steps, and hence the algorithm satisfies bounded exit. □

Theorem 3.11. *If the SpecialSet and ticket dispenser implementations used by the algorithm in Figure 3.1 satisfy Specifications 1, 2, 3, 4, and 5, then the algorithm satisfies FCFS, lockout freedom, mutual exclusion, and bounded exit.*

Proof. Follows from Lemma 3.6, 3.8, 3.9, and 3.10. □

3.4.2 Correctness When Using Conditional *SpecialSet* and Ticket Dispenser implementations

In this section we prove that the algorithm in Figure 3.1 is correct when it uses the *SpecialSet* and ticket dispenser implementations presented in [13]. We do this by showing that processes invoke the *SpecialSet* and ticket dispenser according to Conditions 1, 2, 3, and 4. This implies that the *SpecialSet* and ticket dispenser implementations presented in [13] behave correctly when used in our algorithm. Thus, in conjunction with Theorem 3.11, we conclude that the FCFS mutual exclusion algorithm that uses these *SpecialSet* and ticket dispenser implementations is also correct.

3.4.2.1 *SpecialSet* Behaves Correctly

Below we prove that the algorithm uses the *SpecialSet* object according to Condition 1 and 2. This allows us to conclude that the implementation presented in [13] satisfies Specifications 1 and 2 when used by our algorithm.

Lemma 3.12. *(Condition 1 and 2) Processes invoke* InsertSelf() *and* RemoveSelf() *in the algorithm in Figure 3.1 according to the following rules:*

1. *The calls to* InsertSelf() *and* RemoveSelf() *made by any process occur in an alternating sequence, beginning with* InsertSelf()*; and*

2. *Operation* RemoveSelf() *is executed in mutual exclusion.*

Proof. The first part follows easily by inspection of the algorithm in Figure 3.1, and the second part follows from the fact that RemoveSelf() is surrounded by the trying and exit protocol of the auxiliary mutual exclusion algorithm at line 5 and line 14. □

Lemma 3.13. *The SpecialSet implementation presented in [13] satisfies Specifications 1 and 2 when used by the algorithm in Figure 3.1.*

Proof. The result follows by Lemma 3.12, Theorem 3.1, and Theorem 3.2.

□

3.4.2.2 Ticket Dispenser Behaves Correctly

Below we prove that Condition 3 and Condition 4 hold. These results are used by Lemma 3.19 to conclude that the ticket dispenser implementation presented in [13] satisfies Specifications 3, 4, and 5 when used by our algorithm.

Lemma 3.14. *(Condition 3) The calls to* ObtainTicket() *and* DoneWithTicket() *made by any process in the algorithm in Figure 3.1 occur in an alternating sequence, beginning with* ObtainTicket().

Proof. The lemma follows easily by inspection of the algorithm in Figure 3.1.

□

Lemma 3.15. *(Condition 4 (a)) Function* DONEWITHTICKET() *is executed in mutual exclusion in the algorithm in Figure 3.1.*

Proof. The lemma follows from the fact that the only call to DONEWITHTICKET() at line 19 is surrounded by the trying and exit protocol of the auxiliary mutual exclusion algorithm at line 16 and line 23.

□

We use the following lemma in the proof that Condition 4 (b) and (c) hold.

Lemma 3.16. *Let H be any history generated by the algorithm in Figure 3.1, and let i_1 and i_2 be indexes, where $i_1 \leq i_2$. If a process p executes continuously inside* OBTAINTICKET() *during $H[i_1, i_2]$, then no process q that enters the waiting room at some step $i_q \in [i_1, i_2]$ (i.e., q reaches line 5 at step i_q), enters the CS in $H[i_q, i_2]$.*

Proof. Assume that a process p starts OBTAINTICKET() before $H[i_1]$, and next finishes OBTAINTICKET() after $H[i_2]$, i.e., p is continuously in OBTAINTICKET() during $H[i_1, i_2]$. Process p will have finished its last call to *Set*.INSERTSELF() (line 3) before $H[i_1]$ but not started a call to *Set*.REMOVESELF() (line 7) before $H[i_2]$. By Lemma 3.13, the *SpecialSet* used by the algorithm satisfies Specification 1, and so any call by some other process to *Set*.REMOVESELF() at line 7 that starts after $H[i_1]$ and finishes before $H[i_2]$ (i.e., the call to *Set*.REMOVESELF() occurs entirely during $H[i_1, i_2]$) must return a value different from $\perp$.

Suppose, by way of contradiction, that there exists a process q that enters the waiting room at some step $i_q \in [i_1, i_2]$ and enters the CS in $H[i_q, i_2]$. This implies that q fully executes *Set*.REMOVESELF() (at line 7) in $H[i_q, i_2]$. By the argument in the preceding paragraph, q's invocation of *Set*.REMOVESELF() must return a value $v \neq \perp$. Process q

subsequently adds v to the priority queue with a dummy ticket at line 9. Let i'_q be an index such that $H[i'_q]$ is the state immediately after q does this. At state $H[i'_q]$, a dummy ticket exists in the priority queue. We claim that this holds for every subsequent state until after p finishes executing ObtainTicket():

Claim 3.16.1. *A process with a dummy ticket exists in the priority queue in every state in $H[i'_q, i_2]$.*

Proof. Suppose, by way of contradiction, that there is an index $i_s \in [i'_q, i_2]$ such that no dummy ticket is in the priority queue in state $H[i_s]$, and assume $H[i_s]$ is the first such state. The step immediately before this state must have been taken by some process r, which removes the last dummy ticket in the priority queue at line 10. Process r's last execution of *Set*.RemoveSelf() at line 7 prior to this must have started after $H[i'_q]$ and ended before $H[i_s]$, and hence occurred entirely during $H[i_1, i_2]$. This and the argument in the first paragraph of the proof of this lemma imply that r's last execution of *Set*.RemoveSelf() returned some value $r' \neq \perp$, which is added by r to the priority queue with a dummy ticket at line 9.

By Specification 1 and Lemma 3.13, the call to *Set*.RemoveSelf() by r does not return r, and so $r \neq r'$. This implies that the entry that r added to the priority queue at line 9 is different from the entry that r removed from the priority queue at line 10. In turn, this implies that a dummy ticket exists in the priority queue after r executes line 10. This contradicts that there is no dummy ticket in the priority queue after r executes line 10.

□

After q adds v to the priority queue with a dummy ticket, it removes any dummy ticket associated with itself at line 10. At this point, q is no longer in *Set*, and so by Lemma 3.13 and Specification 1, any call to *Set*.RemoveSelf() cannot return q until q starts *Set*.InsertSelf() in its next passage (if any). This implies that after q removes

its dummy ticket at line 10, no process adds q to the priority queue with a dummy ticket until after q executes through the CS.

By the assumption that q enters the CS in $H[i_q, i_2]$, q must read $Head[q] = \mathsf{true}$ at line 15 in $H[i'_q, i_2]$. Process q sets $Head[q] = \mathsf{false}$ at line 6, and so some process r must set $Head[q] = \mathsf{true}$ after this. There are two places where this can occur: either at line 13 or at line 22. Both of these lines are protected by a lock that q occupies when it executes line 6, and so the earliest r sets $Head[q] = \mathsf{true}$ is after q executes line 10. Whichever line r executes to set $Head[q] = \mathsf{true}$ (line 13 or line 22), r has to discover q as the head of the priority queue in its preceding call to Q.FindMin() (line 12 or line 20, respectively). This call also occurs after q executes line 10. By Claim 3.16.1, a dummy ticket exists in the priority queue in every state in $H[i'_q, i_2]$, and so r's call to Q.FindMin() returns the id of a process that has a dummy ticket in the priority queue. However, as r's call to Q.FindMin() occurs after q executes line 10, it follows from the preceding paragraph that process q is not in the priority queue with a dummy ticket during r's call to Q.FindMin(). Thus r's call to Q.FindMin() does not return q, which is a contradiction.

□

The following lemma is a slightly stronger version of Condition 4 (b), which says that if processes p and q are participating simultaneously and hold tickets t_p and t_q, respectively, where $t_p \lhd t_q$, then q does not *subsequently* call DoneWithTicket() before p. We omit the word "subsequently" in the following lemma.

Lemma 3.17. *(Condition 4 (b)) Suppose processes p and q each execute a passage of the algorithm in Figure 3.1. If processes p and q are participating simultaneously and hold tickets t_p and t_q, respectively, where $t_p \lhd t_q$, then q does not call* DoneWithTicket() *before p.*

Proof. Let H be any history generated by the algorithm in Figure 3.1. We prove the

following statement, from which the lemma will follow: For any $j \geq 0$, Condition 4 (b) holds in $H[0, j]$. The proof is by induction on j. In the base case $j = 0$. No processes take any steps in $H[0, 0]$, and so Condition 4 (b) trivially holds in $H[0, 0]$. For the induction hypothesis, let $j \geq 0$ and assume that Condition 4 (b) holds in $H[0, j]$. This, along with Lemmas 3.14, 3.15, and 3.18, imply that Condition 3 and all parts of Condition 4 hold in $H[0, j]$. In turn, this, Theorem 3.3 and Theorem 3.4 imply that the ticket dispenser implementation presented in [13] also satisfies Specifications 3, 4, and 5 in $H[0, j]$. If H is finite and ends in state $H[j]$ (or earlier), then we're done. So assume that H does not end at state $H[j]$.

For the induction step, suppose, by way of contradiction, that Condition 4 (b) is not true in $H[0, j+1]$. By the induction hypothesis, Condition 4 (b) is true in $H[0, j]$, and so the j'th step in the execution is the first step in the execution after which the condition is violated. Consequently, the following facts are true: (i) there exist processes p and q that simultaneously hold tickets t_p and t_q, respectively, in state $H[j]$, (ii) $t_p \lhd t_q$, (iii) p is not executing DoneWithTicket() in state $H[j]$, and (iv) in step j process q begins executing DoneWithTicket() at line 19.

Recall that we established in the first paragraph that the ticket dispenser satisfies Specifications 3, 4, and 5 in $H[0, j]$, and by Lemma 3.13, the *SpecialSet* implementation satisfies Specifications 1 and 2 throughout H, and hence $H[0, j]$. This, facts (i) and (ii) above, and Specification 4 (with the roles of p and q interchanged, stated in the contrapositive), imply that p starts its last execution of ObtainTicket() in $H[0, j]$ before q finishes its last execution of ObtainTicket() in $H[0, j]$. There are two cases to consider: either q enters the waiting room after p finishes ObtainTicket(), or before p finishes ObtainTicket(). If q enters the waiting room after p finishes ObtainTicket(), then clearly q does not enter the CS before p finishes ObtainTicket(). In the latter case, if q enters the waiting room before p finishes ObtainTicket(), i.e., while p is executing inside ObtainTicket(), then by Lemma 3.16, q does not enter the CS before

p finishes OBTAINTICKET(). Hence, in either case, in $H[0, j]$, q does not enter the CS before p finishes OBTAINTICKET().

In state $H[j]$ p has not yet entered the CS — otherwise, because of the lock protecting lines 17–22, and the fact that q begins executing DONEWITHTICKET() at line 19 in step j (fact (iv) above), it must be the case that p finished DONEWITHTICKET() prior to state $H[j]$, contradicting the assumption that q calls DONEWITHTICKET() before p. Furthermore, q does not enter the CS before p finishes OBTAINTICKET(), as shown in the preceding paragraph. Therefore, at the earliest state after both p and q have finished OBTAINTICKET(), p and q simultaneously hold their tickets t_p and t_q, and neither process is in the CS. So, by the STFS property (Lemma 3.5), and the fact that $t_p \lhd t_q$, it follows that q does not enter the CS before p. By inspection of algorithm, we see that the CS and DONEWITHTICKET() are part of the same locked segment of code. Therefore, we conclude that q does not start executing DONEWITHTICKET() in the j'th step unless p finishes executing DONEWITHTICKET() prior to the j'th step. This and fact (iv) above contradict the fact that q calls DONEWITHTICKET() before p. □

The following lemma is a slightly stronger version of Condition 4 (c), which says that if processes p and q are simultaneously inside OBTAINTICKET(), then q does not subsequently call DONEWITHTICKET() before p completes OBTAINTICKET(). We omit the word "subsequently" in the following lemma.

Lemma 3.18. *(Condition 4 (c)) Suppose process p and q each execute a passage of the algorithm in Figure 3.1. If processes p and q are simultaneously inside* OBTAINTICKET(), *then q does not call* DONEWITHTICKET() *before p completes* OBTAINTICKET().

Proof. Suppose process p and q participate simultaneously inside OBTAINTICKET() at line 4. By Lemma 3.16, neither process can execute through the CS in its current passage

until both processes have finished executing ObtainTicket() in their current passages. This, and the fact that a process does not invoke DoneWithTicket() at line 19 until after it executes through the CS, imply that q does not call DoneWithTicket() before p completes ObtainTicket(). □

Lemma 3.19. *The ticket dispenser implementation presented in [13] satisfies Specifications 3, 4, and 5 when used by the algorithm in Figure 3.1.*

Proof. Lemmas 3.14, 3.15, 3.17, and 3.18, imply that Condition 3 and all parts of Condition 4 are satisfied when the ticket dispenser presented in [13] is used by our algorithm. Hence, by Theorem 3.3 and Theorem 3.4, the ticket dispenser implementation presented in [13] satisfies Specifications 3, 4, and 5 when used by our algorithm. □

Theorem 3.20. *The FCFS ME algorithm in Figure 3.1, in conjunction with the SpecialSet and ticket dispenser implementations presented in [13], satisfies FCFS, lockout freedom, mutual exclusion, and bounded exit.*

Proof. By Lemma 3.13, the *SpecialSet* implementation presented in [13] satisfies Specifications 1 and 2, and by Lemma 3.19, the ticket dispenser implementation presented in [13] satisfies Specifications 3, 4, and 5. This and Theorem 3.11 imply the result. □

3.4.3 RMR Complexity

Theorem 3.21. *The RMR complexity of the mutual exclusion algorithm in Figure 3.1, when used in conjunction with the SpecialSet and ticket dispenser presented in [13], is* $O(\min(c, \log N))$ *in both the CC and DSM models, where c denotes point contention.*

Proof. By Lemma 3.13 and 3.19, the *SpecialSet* and the ticket dispenser presented in [13] satisfy Specifications 2 and 5 when used by ME algorithm in Figure 3.1. This implies that the step complexity of the *SpecialSet* and ticket dispenser are both $O(\min(c', \log N))$,

where c' is the maximum number of processes participating simultaneously. The parameter c' is at most c, the point contention, and so the step complexity (hence RMR complexity) of the *SpecialSet* and ticket dispenser is $O(\min(c, \log N))$. The auxiliary mutual exclusion algorithm used at lines 5, 14, 16, and 23 has RMR complexity $O(\min(c, \log N))$ provided that an algorithm such as Kim and Anderson's is used [30]. The priority queue Q is accessed sequentially, and so it is possible to implement with step complexity $O(\log c)$ using elementary data structure techniques.

The busy-wait loop at line 15 must be analyzed separately for the CC and DSM models. In the DSM model, this loop incurs zero RMRs because process p only accesses $Head[p]$, which is allocated in p's local memory module. In the CC model, this loop costs at most two RMRs. The first one occurs when p loads $Head[p]$ into its local cache upon writing $Head[p] :=$ false at line 6. The second one occurs as soon as another process writes $Head[p]$, and then p reads the value written. Following the second RMR, p breaks out of the busy-wait loop because the last write to $Head[p]$ must have occurred at line 13 or 22, and assigned the value true.

Finally, the remaining lines of the ME algorithm complete in $O(1)$ steps. Thus, the ME algorithm has RMR complexity $O(\min(c, \log N))$ per passage, as wanted. □

Chapter 4

FCFS Abortable ME and Group ME

In this chapter we present a transformation that converts an arbitrary abortable ME algorithm into an FCFS abortable ME algorithm. The transformation has the following properties: it uses only reads and writes, and makes $O(N + f(N))$ RMRs in both the CC and DSM models, where $f(N)$ is the RMR complexity of the abortable ME algorithm. This transformation, in conjunction with Lee's abortable ME algorithm [15], yields the first known FCFS abortable mutual exclusion algorithm that uses only reads and writes.

4.1 High-level Description

The transformation is presented in Figure 4.1. The trying, exit, and abort protocols of the abortable ME algorithm used in the transformation are denoted by MutexTrying(), MutexExit(), and MutexAbort(), respectively. We assume that the abortable ME algorithm satisfies mutual exclusion, deadlock freedom, bounded exit, and bounded abort, and has RMR complexity $f(N)$. The transformation defines one abort point at line 33, at which a process can choose to start executing the abort protocol, and we also assume at least one additional abort point is defined by the abortable ME algorithm inside MutexTrying(). (Note that we label line 34 as an abort point, although this is only

Figure 4.1: Transformation to FCFS abortable ME for process $p \in \{1, ..., N\}$

shared variables:
Request: **array**[1..N] **of boolean** *init false*
Barricade: **array**[1..N][1..N] **of boolean**
(In the DSM model, *Barricade*[p][j] and *Request*[p] are local to process p $\forall j$)

private variables:
predecessor_set: **set of integer**

```
25 loop
26 |  NCS
27 |  predecessor_set := ∅                                   // Doorway begins.
28 |  foreach j ∈ {1..N} do
29 |  |  Barricade[p][j] := true
30 |  |  if Request[j] then predecessor_set := predecessor_set ∪ {j}
31 |  Request[p] := true                                     // Doorway ends.
32 |  foreach j ∈ predecessor_set do
33 |  |  await ¬Barricade[p][j]                             // Abort point.
34 |  MutexTrying()                                         // Abort point.
35 |  CS
36 |  MutexExit()
37 |  Request[p] := false
38 |  for j ∈ {1..N} do
39 |  |  Barricade[j][p] := false
40 end loop
```

abort protocol:

```
41 if aborted at an abort point of MutexTrying() then
42 |  MutexAbort()
43 Request[p] := false
44 for j ∈ {1..N} do
45 |  Barricade[j][p] := false
```

to signify that there is an abort point within MutexTrying(). In particular, we do not mean that a process can abort at an arbitrary point in MutexTrying().)

The transformation works as follows. When a process p leaves the NCS, it starts by executing the doorway, which consists of lines 27..31. Process p first determines the set of processes that must precede it when entering the CS (lines 28..30). This is the set of processes that have already completed the doorway but have not yet entered the CS or aborted. As p builds this "predecessor" set, it also enables a barricade variable $Barricade[p][j]$ for every process j (line 29). This variable is local to p in the DSM model and will be used later while waiting for processes in the predecessor set to finish the CS. The final thing that p does in the doorway is to set its request flag, $Request[p]$, to be true, so that other processes know that it has finished the doorway.

The FCFS property states that if a process q completes the doorway before a process p begins the doorway, then p cannot enter the CS before q enters the CS or aborts its entry attempt. To ensure this, process p busy-waits until all the processes in the predecessor set have finished the CS or aborted (lines 32..33).

Any processes that are in the doorway concurrently with p and are not added to p's predecessor set may try to enter the CS at the same time as p. To enforce mutual exclusion in this case, the CS is protected by an abortable mutual exclusion algorithm. After p executes MutexTrying() (line 34), it enters the CS.

After leaving the CS, p executes its exit protocol (lines 36..39). This consists of executing MutexExit() (line 36), and then resetting its request flag to indicate that it has finished the CS (line 37). Finally p scans through all the processes in the system, resetting barricade variables (lines 38..39) so that any processes waiting for p (at line 33) may advance.

The abortability feature of this algorithm allows a process to abort its attempt to enter the CS while waiting at line 33 or while at an abort point in MutexTrying(). The abort protocol is defined in lines 41..45 of Figure 4.1. If a process decides to abort its entry

attempt at line 33, it simply needs to execute lines 43..45. If p aborts its entry attempt while executing MutexTrying(), it additionally needs to execute MutexAbort() (line 42) at the beginning of the abort protocol.

4.2 Proof of Correctness

In this section we prove that the FCFS abortable ME algorithm that results from the transformation is correct and has RMR complexity $O(N + f(N))$.

Lemma 4.1. *The algorithm in Figure 4.1 satisfies the mutual exclusion property.*

Proof. This follows from the fact that the CS is surrounded by trying and exit protocols (MutexTrying(), MutexExit()) of the abortable ME algorithm (lines 34 and 36). □

Lemma 4.2. *The algorithm in Figure 4.1 satisfies the FCFS property.*

Proof. Assume that a process p finishes the doorway before a process q starts it, and that p and q do not abort their entry attempt. Suppose, by way of contradiction, that q enters the CS before p. By this and inspection of the algorithm, it follows that process p sets $Request[p] = \mathsf{true}$ at line 31 before q begins the doorway. Moreover, process p does not set $Request[p] = \mathsf{false}$ until line 37, after the CS. These preceding facts and the assumption that q enters the CS before p imply that $Request[p] = \mathsf{true}$ is invariant between the time q starts the doorway to when q enters the CS. This means that in iteration $j = p$ of the loop at line 28, q reads $Request[p] = \mathsf{true}$ at line 30. Hence, q adds p to its predecessor set at line 30 in the doorway.

In iteration $j = p$ of the loop at line 28, process q sets $Barricade[q][p] = \mathsf{true}$ at line 29. The only process that can set $Barricade[q][p]$ to false is process p, in its exit (or abort) protocol, in the loop at line 38 (or line 44). By this, the assumption that p finishes the doorway before q starts it, and the assumption that q enters the CS before p, it follows that from the time that q sets $Barricade[q][p] = \mathsf{true}$ to the time when p enters the CS, $Barricade[q][p] = \mathsf{true}$ is invariant.

Since q adds p to its predecessor set in the doorway, q busy-waits at line 33 (in iteration $j = p$) until $Barricade[q][p] = \mathsf{false}$. However, we showed in the preceding paragraph that $Barricade[q][p] = \mathsf{true}$ is invariant until p enters the CS. This means that q cannot enter the CS before p, contradicting the assumption that it does. □

Lemma 4.3. *The algorithm in Figure 4.1 satisfies the bounded exit property.*

Proof. By the bounded exit property of the abortable ME algorithm, and inspection of the algorithm in Figure 4.1, it follows that the exit protocol, which consists of lines 36..39, finishes after a bounded number of steps. Therefore, the algorithm satisfies the bounded exit property. □

Lemma 4.4. *The algorithm in Figure 4.1 satisfies the bounded abort property.*

Proof. By inspection, a process can reach line 33, an abort point, in a bounded number of its own steps from anywhere in lines 27..32. Furthermore, by assumption, the abortable ME algorithm satisfies bounded abort. Therefore, the following two facts hold: (i) if a process is in the trying protocol and it cannot enter the CS in a bounded number of its own steps, then it can reach an abort point in a bounded number of its own steps; and (ii) a process can execute MutexAbort() in a bounded number of its own steps. In turn, point (ii) and inspection of the algorithm in Figure 4.1, imply that the abort protocol (lines 41..45) finishes after a bounded number of a process's own steps. Therefore, the algorithm satisfies the bounded abort property. □

Lemma 4.5. *The algorithm in Figure 4.1 satisfies the deadlock freedom property.*

Proof. Suppose, by way of contradiction, that the algorithm does not satisfy deadlock freedom. This implies that there exists an execution history in which a process gets stuck forever in the trying protocol and a point in the history after which no process executes through the CS. By inspection, the only place in the trying protocol where a process can potentially be stuck forever is at line 33, or in MutexTrying().

Claim 4.5.1. *No process can be stuck forever in the method* MutexTrying().

Proof. Suppose, by way of contradiction, some process p is stuck forever in MutexTrying(). By assumption, the abortable ME algorithm satisfies deadlock freedom, which says that if a process is in MutexTrying() then some process eventually enters the CS. This, and our assumption that p is stuck forever in MutexTrying(), implies that there is some process that executes through the CS infinitely often. In turn, this contradicts that there is a point in the history after which no process executes through the CS. □

Claim 4.5.2. *No process can be stuck forever at line 33.*

Proof. Suppose, by way of contradiction, that some process p is stuck forever at line 33. Assume, without loss of generality, that of all the processes that get stuck forever at line 33 in the execution, p is the one that executes line 31 the earliest.

Since process p is stuck forever at line 33, there exists some iteration $j = q$ of the loop at line 32 past which p does not advance. This implies that $q \in p.predecessor\_set$, and so p must have read $Request[q] = \mathsf{true}$ when it last executed line 30 so that it could add q to $p.predecessor\_set$ at line 30.

Let t_1 be the last step in the history at which p executes line 29 (setting $Barricade[p][q] = \mathsf{true}$), and t_2 be the next step after t_1 at which p executes line 30 (reading $Request[q] = \mathsf{true}$).

The only place in the algorithm where $Barricade[p][q]$ is set to true is at line 29 (for iteration $j = q$), and this can only be done by process p. The only place in the algorithm where $Barricade[p][q]$ is set to false is at line 39 (for iteration $j = p$) and this can only be done by process q. Therefore, if q sets $Barricade[p][q] = \mathsf{false}$ at line 39 after t_1 (i.e., the step at which p sets $Barricade[p][q] = \mathsf{true}$ for the last time), then $Barricade[p][q] = \mathsf{false}$ holds from that point until the end of the history. Process p, however, does not advance past iteration $j = q$ of the loop at line 32, and so p reads $Barricade[p][q] = \mathsf{true}$ infinitely

often at line 33. This implies that q cannot execute line 39 at any point after t_1. Step t_2 happens after t_1, and so q cannot execute line 39 at any point after t_2

Furthermore, p reads $Request[q] = \mathsf{true}$ at line 30 (at step t_2), and so it must be the case that $q@\{32..37\}$ holds in the state immediately before t_2. This, combined with the preceding fact that q cannot execute line 39 at any point after t_2, implies that after t_2, q must be stuck forever somewhere in lines 32..37, or in the abort protocol. Process q cannot be stuck forever in the exit protocol or the abort protocol since the algorithm satisfies bounded exit and bounded abort. Process q cannot be stuck forever in the CS since we assume that processes do not crash and that the CS is finite. By claim 4.5.1, q cannot be stuck forever in MutexTrying(). The only remaining place where q can be stuck forever is at line 33. However, since p reads $Request[q] = \mathsf{true}$ at line 30 (at step t_2), this implies that q executes line 31 before p does. This contradicts our assumption that of all the processes that get stuck forever at line 33, p is the one that executes line 31 the earliest. □

Recall that the only place in the trying protocol at which a process can get stuck forever is at line 33, or in MutexTrying(). By Claim 4.5.1 and Claim 4.5.2, however, no process can be stuck forever at these places. Therefore, there exists no process that is stuck forever in the trying protocol, contradicting that there is such a process. □

Lemma 4.6. *The algorithm in Figure 4.1 satisfies the lockout freedom property.*

Proof. This lemma follows immediately from the fact that the algorithm satisfies FCFS (Lemma 4.2) and deadlock freedom (Lemma 4.5). □

Lemma 4.7. *The algorithm in Figure 4.1 has RMR complexity $O(N + f(N))$ in the CC (resp. DSM) model, where $f(N)$ is the RMR complexity of the abortable ME algorithm in the CC (resp. DSM) model.*

Proof. By inspection, there are $O(N)$ RMRs made in the doorway. Furthermore, by inspection, there are $O(N + f(N))$ RMRs made in the exit protocol and abort protocol.

The only part of the algorithm where more than $O(N + f(N))$ RMRs may be made is in lines 32..34. The RMR complexity of the abortable ME algorithm is $f(N)$ in the CC (resp. DSM) model, and so the total number of RMRs made at line 34 is $O(f(N))$. We now show that $O(N)$ RMRs are made in the loop at line 32 in both the CC and DSM models.

We first consider the CC model. The only process that sets $Barricade[p][j] = \mathsf{true}$ is process p at line 29. Therefore, while process p is at line 33, no other process will write the value true into $Barricade[p][j]$. This implies that if p makes two RMRs while waiting on the variable $Barricade[p][j]$, p is guaranteed that the value it reads the second time is false, and p will advance to the next iteration of the loop at line 32. Therefore the total number of RMRs made in the loop at line 32 is $O(N)$.

In the DSM model, $Barricade[p][j]$ is local to process p for all j, and so no RMRs are made at line 33. □

Theorem 4.8. *The algorithm in Figure 4.1 satisfies mutual exclusion, FCFS, bounded exit, bounded abort, and lockout freedom. Furthermore, the algorithm has $O(N + f(N))$ RMR complexity in the CC (resp. DSM) model, where $f(N)$ is the RMR complexity of the abortable ME algorithm.*

Proof. The theorem follows from Lemmas 4.1, 4.2, 4.3, 4.4, 4.6, and 4.7. □

Theorem 4.9. *There exists an FCFS abortable mutual exclusion algorithm that uses only reads and writes, and has $O(N)$ RMR complexity.*

Proof. Lee's abortable mutual exclusion algorithm [15, 35] uses only reads and writes and has RMR complexity $O(\log N)$ in both the CC and DSM model. If we instantiate the transformation in Figure 4.1 using this algorithm, then by Theorem 4.8, we arrive at an algorithm with the desired properties. □

4.3 Group Mutual Exclusion

By Theorem 4.9 there exists FCFS abortable mutual exclusion algorithm that has $O(N)$ RMR complexity in both CC and DSM models, and uses only reads and writes. This algorithm can be used as a building block for a new local-spin group mutual exclusion algorithm that also uses only reads and writes. This is done using the transformation presented by Danek and Hadzilacos [14] for the DSM model. By using the FCFS abortable mutual exclusion algorithm from this chapter in that transformation, we end up with the first local-spin group mutual exclusion algorithm that uses only atomic reads and writes. The resulting GME algorithm has $O(N)$ RMR complexity in the DSM model, which is asymptotically optimal [14]. We summarize these results in the following theorem.

Theorem 4.10. *There exists a group mutual exclusion algorithm that uses only reads and writes, and has $O(N)$ RMR complexity in the DSM model.*

Note that the preceding result also applies to the CC model, although the transformation in [14] was only studied in the DSM model.

Chapter 5

The k-Bakery

In this chapter we present the first known local-spin k-exclusion algorithms that use only atomic reads and writes. All algorithms, with the exception of one, have $O(N)$ RMR complexity in both the CC and DSM models, and the high-level structure of all the algorithms is inspired by Lamport's famous Bakery algorithm. We additionally present local-spin k-exclusion algorithms, also having $O(N)$ RMR complexity in both the CC and DSM models, that require only the use of non-atomic reads and writes.

We start by reviewing the Bakery algorithm briefly, as its structure is the basis for the algorithms in this chapter. After this, we provide a conceptual overview of our different k-exclusion algorithms, after which we describe the asynchronous shared-memory model for non-atomic reads and writes. We then present a sequence of k-exclusion algorithms, starting with one that is conceptually very simple, gradually presenting more complex and powerful algorithms as we progress through the sequence, and ending with a k-exclusion algorithm that is conceptually the most complex but also the most "feature-rich" in terms of the set of properties that it satisfies. We close this chapter by summarizing the different techniques and mechanisms used by our k-exclusion algorithms.

5.1 Lamport's Bakery Algorithm

Lamport's Bakery algorithm [32] is a non-local-spin FCFS mutual exclusion algorithm that is correct even when reads and writes are non-atomic. It is given in Figure 5.1. The name of the algorithm arises from the fact that processes behave like people waiting in line at a bakery. Each process chooses a ticket and then waits for its turn to be served by the CS.

The algorithm uses two shared arrays: *Doorway*, and *Ticket*. A process p sets $Doorway[p]$ to be true at line 48 to indicate to other processes that it has started the doorway, and then sets it to false at line 50 when it finishes the doorway. At line 49 of the doorway, process p chooses a ticket used to indicate its order of priority to enter the CS. In the waiting room, process p waits for each other process q to finish the doorway (line 52), and then waits until it has priority over q to enter the CS (line 53). Process p has priority over q to enter the CS if either q is not requesting entry into the CS (indicated by $Ticket[q] = 0$) or q's ticket is larger than p's ticket (using process ids to break ties between equal tickets). When a process finishes the CS, it resets its ticket to 0, indicating that it is returning to the NCS. Note that despite the fact that a process resets its ticket to 0 in the exit protocol, tickets can grow without bound in this algorithm.

5.2 Conceptual Overview of k-Bakery Algorithms

The high-level structure of our k-exclusion algorithms is given in Figure 5.2. In the doorway, a process first acquires a ticket that is larger than any tickets currently held by other processes. It then waits until there are fewer than k processes with smaller tickets. Once this is the case, the process can safely enter the CS. Finally, the process discards its ticket in the exit protocol. Notice that this is nearly the same as the high-level structure of Lamport's Bakery algorithm. The only difference is that on line 60,

Figure 5.1: Lamport's Bakery algorithm for process $p \in \{1, ..., N\}$

```
shared variables:
    Doorway: array[1..N] of boolean init all false
    Ticket: array[1..N] of ℕ init all 0

46 loop
47 |  NCS
48 |  Doorway[p] := true
49 |  Ticket[p] := 1 + max(Ticket[1], Ticket[2], ..., Ticket[N])
50 |  Doorway[p] := false
51 |  for i := 1 to N do
52 |  |  await Doorway[i] = false
53 |  |  await Ticket[i] = 0 ∨ (Ticket[i], i) ≥ (Ticket[p], p)
54 |  CS
55 |  Ticket[p] := 0
56 end loop
```

in Lamport's Bakery algorithm, instead of waiting until there are fewer than k smaller tickets, a process simply waits until there are no smaller tickets.

Figure 5.2: High-level structure of k-exclusion algorithms

```
57 loop
58 |  NCS
59 |  Acquire the next largest ticket                    // Doorway
60 |  Wait until there are fewer than k smaller tickets
61 |  CS
62 |  Discard ticket
63 end loop
```

There are, of course, many details left undefined by Figure 5.2: How does a process acquire a ticket? How does a process check if there are fewer than k smaller tickets? How does a process discard a ticket? These details will be described in the coming sections where we actually present our algorithms. The goal in this section, however, is to provide a conceptual overview of our algorithms, highlighting their commonalities and their differences.

All of our algorithms satisfy k-exclusion, starvation freedom, bounded exit, and k-

FCFS. They also all have $O(N)$ RMR complexity in both the CC and DSM models, with the exception of the algorithm in Section 5.5, which we discuss in more detail below. The main differences between our algorithms are with respect to the following characteristics:

1. **(Ticket Resetting)** When a process discards a ticket, does it reset its ticket to zero, as in Lamport's Bakery algorithm, or not?

2. **(Synchronization Primitives)** Does the algorithm require atomic reads and writes, or does it work with even non-atomic reads and writes?

3. **(FIFE)** Does an algorithm satisfy the FIFE property?

Ticket resetting deserves further explanation. In the Bakery algorithm, when a process "discards" its ticket, it does so by setting its entry in the *Ticket* array to be 0 (line 55). In contrast, in some of our algorithms, a process discards its ticket in a different manner that does not involve resetting ticket values. This results in algorithms in which tickets grow without bound in an unrestricted manner. In a number of our other algorithms, tickets are reset to 0 in the exit protocol. Just as in Lamport's original Bakery algorithm, tickets can still grow without bound, but this happens only in certain situations where some process is always outside of the NCS. In particular, in the algorithms where tickets are reset, if there is a period of "quiescence" in which all processes are in the NCS, all variables will be reset to their initial values.

Ticket resetting is a desirable feature since it limits the situations under which variables can grow without bound. It turns out, however, that it is more difficult to implement our k-exclusion algorithms with this feature than without. As such, we begin our presentation with simpler algorithms that do not have this feature.

The first k-exclusion algorithm that we present is given in Section 5.4, and is conceptually the simplest. This algorithm does not reset tickets, requires atomic reads and writes, and does not satisfy FIFE. (We temporarily defer a description of the algorithm in Section 5.5.) The algorithm in Section 5.6 additionally satisfies FIFE, and is further

embellished in Section 5.7 to also work with non-atomic reads and writes. In Sections 5.8 - 5.10 we present our ticket-resetting algorithms. We begin with a relatively simple k-exclusion algorithm that resets tickets, uses only non-atomic reads and writes, and does not satisfy FIFE. Modifying this algorithm to satisfy FIFE turns out to require a number of complex mechanisms that are easier to describe if we first present a version of the algorithm that uses atomic reads and writes. In Section 5.9 we present such an algorithm: it resets tickets, uses atomic reads and writes, and satisfies FIFE. Finally, we embellish this algorithm in Section 5.10 to arrive at a k-exclusion algorithm that resets tickets, uses only non-atomic reads and writes, and satisfies FIFE.

The algorithm presented in Section 5.5 is nearly identical to the one given in Section 5.4, except that it is local-spin only in the CC model and is more space-efficient: it requires $\Theta(N)$ shared variables, as opposed to $\Theta(N^2)$ shared variables, which is the space requirement of the other algorithms in this chapter. We did not identify "space-efficiency" above as one of the distinguishing characteristics of the algorithms, since it turns out that the techniques used in all the algorithms in Section 5.6 and later require $\Theta(N^2)$ shared variables.

The preceding discussion is summarized in Table 5.1.

Algorithm	FIFE	NARW[1]	TR[1]	$O(N)$ RMRs (DSM)	$O(N)$ RMRs (CC)
Section 5.4				✓	✓
Section 5.5					✓
Section 5.6	✓			✓	✓
Section 5.7	✓	✓		✓	✓
Section 5.8		✓	✓	✓	✓
Section 5.9	✓		✓	✓	✓
Section 5.10	✓	✓	✓	✓	✓

Table 5.1: Summary of k-exclusion algorithms.

5.3 Asynchronous Shared-Memory Model For Non-atomic Reads and Writes

The model that we use for non-atomic reads and writes is similar to the atomic read and write model described in Chapter 2, except that when a process takes a step, the step is not a single "indivisible" read or write. Instead, a step is either the invocation of a read or write, or the response to a read or write. Intuitively, a read or write takes place during the time between an invocation and its corresponding response, and if a read of some variable overlaps the write of the same variable, then the value returned by the read is arbitrary. Also, as a result of this change, there is no longer a notion of a well-defined *global* state that exists before and after each step, although each process still has a well-defined private state before and after each step. This model has some high-level similarities to mutli-reader single-writer safe registers as defined by Lamport [34], however Lamport did not consider multiple concurrent writers, and our model also

[1]NARW = Non-atomic Reads and Writes; TR = Ticket Resetting

incorporates process crashes.

A *system* consists of a set of N processes and a set of shared variables to which processes have access. Each shared variable in the system has an *initial value* that corresponds to the value it is assigned before being accessed by any processes. A *process* is an automaton that has a *private state*, which is reflected by the values of a set of *private variables* (i.e., variables to which only that process has access). As elsewhere in this thesis, we define processes informally using pseudocode.

The *initial state* of the system consists of the assignment of initial values to shared variables and the private state of each process prior to any process taking any *steps*. A *step* by a process consists of a private computation, along with exactly one of the following: (i) the *invocation* of a read or write operation on a shared variable, (ii) a *response* from a read or write operation on a shared variable, or (iii) a *crashing response* from a read or write operation on a shared variable. We explain why we need to define crashing response steps in addition to (ordinary) response steps below, after we first define how we model executions.

An *execution* (or *execution history*) is modelled by a sequence of process steps. We say that an execution is *valid* if it satisfies certain properties:

- **P1:** The first time a process takes a step, the step is an invocation.
- **P2:** If a step t by a process p is an invocation, and t is not the process's first step, then the last step taken by p before t is a response.
- **P3:** If a step t by a process p is a (possibly crashing) response to a read (resp., write) operation on some shared variable X, then the last step taken by p before t is an invocation of a read (resp., write) operation on X.
- **P4:** If a history H is infinite, and p takes some finite (non-zero) number of steps in H, then the last step taken by p is a response.

- **P5:** If a step t by a process p is a crashing response, then p takes no steps after t.

Intuitively, properties P1, P2 and P3 ensure that the sequence of steps taken by a specific process must be an alternating sequence of invocation and response steps. Property P4 ensures that in any infinite execution, every invocation eventually receives a response (possibly a crashing response), and property P5 ensures that a process takes no steps after a crashing response step.

For an execution to be valid, the values returned by read operations in response steps must also obey certain properties. To define these properties precisely, we first need to define some notation:

For any execution H, and any steps a and b in H, we write $a \to b$ if a happens before b in H. An *operation* A by a process p is a non-empty subsequence of steps in H taken by process p, starting with an invocation, such that if a, b are steps in A, c is a step in H, and $a \to c \to b$, then c is also a step in A. We say that A is an *incomplete operation* if the last step in A is an invocation step, and otherwise say it is a *complete operation*. (For this section, whenever we refer to an operation, we mean one that can be either complete or incomplete. However, in the rest of the chapter, in the proofs of correctness for our algorithms, whenever we refer to an operation, unless otherwise noted, we mean one that is complete.)

For any operations A and B, we write:

- $A \to B$ iff A is finite, the last step $a \in A$ is a response step, and for the first step $b \in B$, $a \to b$
- $A \dashrightarrow B$ iff $B \not\to A$

There are certain properties that hold for the $\to$ and $\dashrightarrow$ relations, which we use throughout the thesis. These properties follow from the above definitions. For any operations A, B, C:

- $\rightarrow$ is irreflexive and transitive, i.e., a partial order.
- If $A \rightarrow B$ then $A \dashrightarrow B$
- If $A \rightarrow B \dashrightarrow C$ or $A \dashrightarrow B \rightarrow C$ then $A \dashrightarrow C$
- if $A \rightarrow B \dashrightarrow C \rightarrow D$ then $A \rightarrow D$.
- if A and B are complete operations by the same process, with no steps in common, then $A \rightarrow B$ or $B \rightarrow A$. (Intuitively, complete non-overlapping operations by the same process can be totally ordered.)

For any operations A, B, we write $A \subseteq B$ iff A and B are operations executed by the same process, and for all steps $a \in A$, it is also true that $a \in B$.

We say that operations are *concurrent* if $A \dashrightarrow B$ and $B \dashrightarrow A$. A *read operation* R on a shared variable X by a process p is an operation consisting of at most two steps: the invocation of the read, and (possibly) its corresponding response. A *write operation* W on a shared variable X is defined analogously. We say that a write operation is a *crashing write* if it contains a crashing response step.

For an execution to be valid, the following properties about read and write operations must be satisfied for every shared variable X in the system:

- **P5:** Let R denote a complete read of X such that for all writes W to X, $R \rightarrow W$. Then R must read the initial value of X. (Intuitively, any read that precedes every write to X must read X's initial value.)
- **P6:** Let R denote a complete read of X such that for all writes W to X, either $W \rightarrow R$ or $R \rightarrow W$. If there exists a write W^* to X such that
  1. W^* is not a crashing write, and
  2. $W^* \rightarrow R$, and
  3. for all writes $W' \neq W^*$ to X, either $R \rightarrow W'$ or $W' \rightarrow W^*$

then R must read the value written by W^*. (Intuitively, any complete read of X that is not concurrent with a write to X must read the value written by the "latest" write to X, unless the latest write to X was concurrent with some other write to X or was a crashing write.)

- **P7:** Let R and R' denote any distinct complete reads of X such that $R \to R'$ and for every write W on X either $W \to R$ or $R' \to W$, then R and R' return the same value. (Intuitively, complete reads that are not concurrent with any write and that do not have any write between them must return the same value.)

The reason for defining a crashing response step in addition to an ordinary response step can now be explained. Intuitively, processes may crash during an execution. We say that p *crashes* in an infinite execution H if p stops taking steps in H while outside the NCS. Property P4 guarantees that in any valid infinite execution, even in one in which processes crash, after a process takes an invocation step, there will be a corresponding response. This is somewhat counterintuitive to the notion of a process crashing: we expect that when a process stops taking steps (and hence crashes) that the last step can be either an invocation or response. Suppose for the moment that this is the case, and that the last step taken by a process in an infinite execution could be an invocation. In this case, defining P6 becomes problematic. In P6, we want to be able to say that a read operation on variable X that is not concurrent with any writes to X will read the value "last written" to X unless the latest write to X was concurrent with some other write to X. If the last step by a process before it crashes is the invocation of a write on a variable X, then that write will appear to other processes as a write that starts but never finishes, and thus will be concurrent with every subsequent read and write of X. As a result, a process crashing may interfere with other operations indefinitely. Again, this is counterintuitive: after a process crashes, it should not be able to interfere with other operations indefinitely. To resolve this, we use the notion of crashing responses.

This immediately solves the problem of incomplete operations interfering with other operations indefinitely. Also, it allows us to model the intuition that if a process crashes after invoking an operation, it should not be forced to "properly" complete the operation.

Lastly, for an execution to be valid, the execution must also satisfy the following property:

- **P8:** The sequence of steps taken by a process p, and the private state of p before and after each step, must be consistent with the automaton for process p, which is defined informally using pseudocode.

Property P8 can be made more precise as follows. For any history H and any process p, let $H|p = t_0, t_1, t_2, ...$ be the subsequence of steps in H taken by p, where t_j is the j'th step in this subsequence. The private state of p immediately preceding t_0 is p's initial private state. Property P8 says that for each $j \geq 0$, step t_j depends on the private state of p immediately preceding step t_j and the pseudocode for p, and that the private state of p immediately following step t_j must be updated according to t_j. In particular, if t_j involves any private computation, then the private state of p is updated according to that private computation. If t_j is an invocation, then the private state of p after t_j must be such that the next allowable step by p after t_j is the response step that corresponds to the invocation made in t_j. If t_j is a response step to a read, then the value returned may (possibly) be assigned to one of p's private variables in the private state of p after t_j. (Property P8 is essentially the analogue of validity as defined for the atomic model, except that in the non-atomic model there is no global state as there is in the atomic model.)

Henceforth, we assume that all executions that we are dealing with are valid. Also, we assume that processes execute *asynchronously*, meaning that between any two steps of some process in an execution history, there can be an arbitrary number of steps taken by other processes.

An *algorithm* is a way of specifying the system (i.e., it is a specification of the program for each process in the system, and each variable's initial value). An "execution of an algorithm" or an "execution generated by an algorithm" is shorthand for saying an execution of the system specified by the algorithm.

We say that a process i *enters* the CS when it takes its first invocation step in the CS. We say that an operation A by a process i in a history H *occurs as part of* the CS if each step in A is taken in the CS. We use similar terminology by substituting "CS" in the preceding with TP, EP, NCS, doorway, or waiting room.

Much of our discussion on correctness properties in Section 2.2 carries over to this model. More precise statements for the correctness properties of k-exclusion are as follows:

k-exclusion : $\mathbb{KE}(H)$: For any $k+1$ operations A_i (for $i = 1..k+1$) by distinct processes in H, if each A_i occurs as part of the CS, then for at least one of these operations, say A_j, $A_j \rightarrow A_i$ or $A_i \rightarrow A_j$ (for all $i \in \{1..N\} \setminus \{j\}$).

Starvation Freedom: $\mathbb{SF}(H)$: If H is infinite, at most $k-1$ processes crash[1] in H, there is an operation A by a process p in H that occurs as part of the TP, and p doesn't crash in H, then there exists an operation B by p in H that occurs as part of the CS and $A \rightarrow B$.

As discussed before, to avoid complexity in our proofs, when proving an algorithm correct we generally use the less formal statements of its correctness properties with the understanding that they have more precise counterparts of the style described above.

Also, we adopt the $\rightarrow$ and $\dashrightarrow$ notation in proofs of algorithms in this chapter that are intended for the atomic read and write model. The semantics of $\rightarrow$ and $\dashrightarrow$ in the atomic read and write model from Chapter 2 carry over naturally from the non-atomic model. More precisely, given a history H in the atomic model, if t_i is the i'th step in H,

[1]Recall, we only count processes that crash outside of the NCS.

and t_j is the j'th step in H, where $j > i$, then $t_i \rightarrow t_j$. An operation A by a process is a non-empty contiguous subsequence of steps by the process in H. The relations $\rightarrow$ and $\dashrightarrow$ are defined for operations in the atomic model as follows. For any operations A, B, we write $A \rightarrow B$ iff A is finite, and for the last step $a \in A$ and the first step $b \in B$, $a \rightarrow b$. We write $A \dashrightarrow B$ iff $B \not\rightarrow A$. The properties of the $\rightarrow$ and $\dashrightarrow$ relations, as listed earlier for the non-atomic model, also hold in the atomic model.

5.3.1 RMRs in the Non-atomic Model

An RMR occurs when a process accesses the processor-to-memory interconnect. This notion is made more precise by considering two types of shared-memory models: the DSM model, and the CC model, both of which we defined earlier. As part of the description of these models, we defined how and when an RMR occurs. The description that we provided, however, was in the context of atomic reads and writes. We now revise these descriptions in the context of non-atomic reads and writes.

It turns out that the description of the DSM model can remain unchanged. A process makes an RMR whenever it accesses a shared variable that is stored in another process's memory module. This is not affected by the fact that reads and writes are no longer atomic. The description of the CC model, however, does require some clarification.

In our earlier description of the CC model, a process p makes an RMR whenever p reads a shared variable for the first time, p writes a shared variable, or p reads a shared variable for the first time after another process writes the same variable. This is poorly defined when using non-atomic reads and writes, as it is not clear whether a read operation of some variable by a process will make an RMR as a result of a concurrent write operation to the same variable. We clarify this issue as follows.

Let R be a read operation of a shared variable X by a process p, and let $\bar{R}$ be the set of all read operations R' of X by p such that $R' \rightarrow R$, i.e., $\bar{R} = \{R' : R' \rightarrow R\}$. Furthermore, let W, if it exists, be the last write to X that finishes before R starts (i.e.,

the response step of W is the last response step of a write to X that appears in the history before the invocation step of R). We say that R *makes an RMR* iff one of the following is true:

1. $\bar{R}$ is empty; or,

2. W exists, and there does not exist $R' \in \bar{R}$ such that $W \rightarrow R'$.

Intuitively, R makes an RMR if R is the first time p reads X, or if R is the first time that p reads X after another process has *finished* writing X. This way of defining RMRs in the non-atomic CC model may seem counterintuitive: a write operation should be able to invalidate cached copies of a variable at any point between its invocation and response, thereby causing concurrent reads of the same variable to make an RMR. In fact, if one interprets cache invalidation as an "operation" that takes place non-atomically, then it could be the case that for a single write of X, every read of X that is concurrent with this write makes an RMR. Processes execute asynchronously in our model, and so there are an unbounded number of operations by other processes between the invocation and response of any write. This means that a single write can cause some other process to make an unbounded number of RMRs. Using these semantics, it is clearly impossible for any algorithm that uses non-atomic reads and writes to be local-spin in the CC model.

Another approach to defining RMRs in the CC model that may seem more reasonable is as follows. Let W be a write operation of X, and let $\hat{R}$ be the set of reads of X by a process p that are concurrent with W (i.e., for each $R \in \hat{R}, R \dashrightarrow W$ and $W \dashrightarrow R$). Suppose we allow some constant number of reads $R \in \hat{R}$ to make an RMR. Intuitively, this behaviour is more appealing than the definition we gave above, since it allows for a write operation to invalidate cached copies of a variable some constant number of times between its invocation and response. It turns out, however, that this definition does not make a difference in terms of our RMR complexity results: Whether we use our more restrictive definition, or if we allow a fixed number of cache invalidations to occur

"during" a write operation, the RMR complexity of our algorithms is asymptotically the same. This can easily be seen by observing that the definition of RMRs in this paragraph, versus the restrictive definition above, increases the number of RMRs that a process can make in a passage only by a constant factor. We thus use the more restrictive definition of RMRs given above.

5.4 Local-spin k-Exclusion (Atomic RWs)

In Figure 5.3 we present a k-exclusion algorithm that has $O(N)$ RMR complexity in the CC and DSM models. The high-level structure of the algorithm is similar to Lamport's Bakery algorithm: processes choose a ticket in the first part of the trying protocol, and then wait for processes with lower-numbered tickets to execute through the CS. There are two main differences between our algorithm and Lamport's algorithm. Firstly, processes announce their ticket at the start of the trying protocol differently. Secondly, in our algorithm a process does not have to wait for every process with a lower-numbered ticket to finish the CS, but rather only waits until there are fewer than k processes with lower-numbered tickets (line 70). We now explain the algorithm in more detail.

Processes use two shared arrays to communicate with each other: the *Want* array, and the *Ticket* array. The *Want* array is used by processes to announce their wish to enter the CS. In particular, the value stored in entry $Want[p][q]$ is a "ticket value", and it indicates to process q whether p is trying to enter the CS. When a process p is in the NCS, $Want[p][q] = \infty$, indicating that p does not want to enter the CS. When $Want[p][q] = v$ for some value $v \neq \infty$, then process p is outside of the NCS and is trying to enter the CS with the ticket value v. Ticket values provide a rough guideline for the order in which processes are admitted to the CS. Ticket values are stored and generated using the shared array *Ticket*. We now explain how these shared arrays are employed in the trying and exit protocols of the algorithm.

The trying protocol (lines 66..72) consists of two parts: the doorway, and the waiting room. The doorway consists of lines 66..67. In this part of the trying protocol, a process first announces itself at line 66 to all other processes with the ticket it chose in its previous passage (0 if it executed no previous passage). A process then chooses a new ticket at line 67. Once a process is done the doorway, it announces the ticket it just chose to all other processes at line 68.

One may wonder why it is not sufficient to simply start the trying protocol by choosing a new ticket (line 67) and then announce it. This is due to a race condition that arises otherwise, but we postpone discussion of this until after we explain the next part of the algorithm.

After announcing its ticket, a process p initializes *predecessor_set* to be the set of all other processes in the system (line 69). The predecessor set is intended to approximate the set of processes that are trying to enter the CS concurrently with p and that have priority over p to enter the CS. We say that a process q has priority over p if q and p choose tickets t_q and t_p and $(t_q, q) < (t_p, p)$.

The purpose of lines 70..72 is to prevent a process p from entering the CS until enough processes are eliminated from p's predecessor set. To this end, p repeatedly checks the state of all other processes still in its predecessor set in the loop at line 71. Once p detects that a process q no longer has priority over it, p removes q from its predecessor set at line 72.

When the size of p's predecessor set goes below k, p enters the CS.

In the exit protocol (line 74), a process simply announces to all other processes that it is returning to the NCS by setting the appropriate values in the *Want* array to ∞.

We now explain the race condition that arises if line 66 is removed from the algorithm. If two processes p and q execute line 67 concurrently, the following may occur: q chooses a smaller ticket than p, but does not yet announce the ticket at line 68. Process p then races ahead of q into the waiting room. As q has not yet announced its ticket to p, p

will see that $Want[q][p] = \infty$, and so p will remove q from its predecessor set. This is premature on p's part, since q actually has priority over p. When q executes the waiting room, it also removes p from its predecessor set, since p has a larger ticket than q. It is easy to see how this can lead to a violation of k-exclusion for $k = 1$. More elaborate execution scenarios can be constructed in which k-exclusion is violated for $k > 1$.

The preceding scenario is avoided if q first announces the ticket it chose in its previous passage. Since q, in its current passage, chooses a ticket smaller than p, the ticket q announces at line 66 is also guaranteed to be smaller than p's ticket. Moreover, by the time p starts the waiting room, q will have finished line 66, otherwise q will choose a ticket larger than p. Thus, when p executes the waiting room, it will not remove q from its predecessor set prematurely.

Figure 5.3: (Atomic RWs) k-Exclusion algorithm for process $p \in \{1, ..., N\}$

```
shared variables:
    Want: array[1..N][1..N] of ℕ ∪ {∞} init all ∞
    Ticket: array[1..N] of ℕ init all 0
    (DSM model: Ticket[p], Want[i][p], are local to process p for all i)
private variables:
    predecessor_set: Set of ℕ

64 loop
65     NCS
66     foreach i ∈ {1..N} \ {p} do  Want[p][i] := Ticket[p]
67     Ticket[p] := 1 + max(Ticket[1], Ticket[2], ..., Ticket[N])
68     foreach i ∈ {1..N} \ {p} do  Want[p][i] := Ticket[p]
69     predecessor_set := {1..N} \ {p}
70     while |predecessor_set| ≥ k do
71         foreach i ∈ predecessor_set do
72             if (Ticket[p], p) < (Want[i][p], i) then
                   predecessor_set := predecessor_set \ {i}
73     CS
74     foreach i ∈ {1..N} do  Want[p][i] := ∞
75 end loop
```

5.4.1 Proofs

Let C_p denote the operation in which process p chooses its ticket at line 67, and let EP_p denote the operation in which p executes its exit protocol at line 74. Note that these operations consist of multiple reads and writes.

Lemma 5.1. *Let p and q be distinct processes. If $C_p \rightarrow C_q$, then $t_p < t_q$, where t_p and t_q are the tickets chosen by p and q, respectively.*

Proof. Process p finishes line 67 before q starts it, and so the value that q reads from $Ticket[p]$ at line 67 is no smaller than t_p. This and inspection of line 67 imply that the ticket t_q that q chooses satisfies $t_q > t_p$. □

As a corollary to the preceding lemma, we state its contrapositive:

Corollary 5.2. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q. If $t_q \leq t_p$, then $C_q \dashrightarrow C_p$.*

Lemma 5.3. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q, respectively. If $(t_q, q) < (t_p, p)$, then p does not remove q from $p.predecessor\_set$ before EP_q starts.*

Proof. Suppose, by way of contradiction, that p removes q from $p.predecessor\_set$ at line 72 before EP_q starts. At line 72, p reads a value w_q in $Want[q][p]$ such that $(t_p, p) < (w_q, q)$ (denote this read operation R) before EP_q starts. Process q writes into $Want[q][p]$ at most three times in its passage: at line 66 it writes the ticket $t'_q < t_q$ that it chose in its preceding passage (denote this write operation W_1); at line 68 it writes t_q (denote this write operation W_2); and finally, at line 74, it writes ∞ (denote this write operation W_3).

Claim 5.3.1. $R \rightarrow W_3$.

Proof. The claim follows from the following two facts: (i) R finishes before EP_q starts, i.e., $R \rightarrow EP_q$ and (ii) $W_3 \subseteq EP_q$. (Claim 5.3.1) □

Claim 5.3.2. $W_1 \rightarrow R$.

Proof. By the assumption that $(t_q, q) < (t_p, p)$, $t_q \leq t_p$. This and Corollary 5.2 imply that $C_q \dashrightarrow C_p$. By inspection of the algorithm, $W_1 \rightarrow C_q$ and $C_p \rightarrow R$. Thus, $W_1 \rightarrow C_q \dashrightarrow C_p \rightarrow R$, and so $W_1 \rightarrow R$. (Claim 5.3.2) □

Since we are using atomic reads and writes, either $R \rightarrow W_2$, or $W_2 \rightarrow R$. We consider these cases separately. In the first case, by Claim 5.3.2, $W_1 \rightarrow R \rightarrow W_2$. W_1 and W_2 are successive writes into $Want[q][p]$, and so R must read the value written by W_1, namely the ticket $t'_q < t_q$ that q chose in its preceding passage. By assumption, $(t_q, q) < (t_p, p)$, which implies that $t_q \leq t_p$; since $t'_q < t_q$, we have $t'_q < t_p$. That is, R must read a value t'_q such that $t'_q < t_p$. This contradicts that R reads a value w_q such that $(t_p, p) < (w_q, q)$, i.e., $w_q \geq t_p$.

In the second case, $W_2 \rightarrow R$. This and Claim 5.3.1 imply that $W_2 \rightarrow R \rightarrow W_3$. W_2 and W_3 are successive writes into $Want[q][p]$ and so the value w_q that R reads must be the value t_q written by W_2, i.e., $w_q = t_q$. By assumption, $(t_q, q) < (t_p, p)$, which contradicts that $(t_p, p) < (w_q, q)$. □

Lemma 5.4. *The algorithm in Figure 5.3 satisfies k-exclusion.*

Proof. Suppose, by way of contradiction, that $k + 1$ processes are in the CS concurrently. Let Y be this set of processes, and let p be the process in Y with the largest $(ticket, process\_id)$ pair. By Lemma 5.3, when p executes the trying protocol, p does not remove any process in Y from $p.predecessor\_set$ before some process in Y starts the exit protocol. This implies that the size of p's predecessor set is at least k until after some process in Y leaves the CS. Thus p cannot enter the CS until after some process in Y leaves the CS, which contradicts that p is in the CS concurrently with all processes in Y. □

Lemma 5.5. *The algorithm in Figure 5.3 satisfies starvation freedom.*

Proof. Suppose, by way of contradiction, that starvation freedom does not hold. Let Y be the set of non-faulty processes that are stuck forever in the TP, and let p be the process in Y with the smallest $(ticket, process\_id)$ pair.

By inspection of the code, we see that p removes from its predecessor set any process that it sees as having announced a larger $(ticket, process\_id)$ pair or as having returned to the NCS. This, the fact that process' tickets increase monotonically, and the assumption that p has the smallest $(ticket, process\_id)$ pair of all the non-faulty processes that get stuck forever, imply that eventually the only processes in p's predecessor set are crashed processes. However, there are at most $k-1$ crashed processes, and so p eventually enters the CS, contradicting that p never enters the CS. □

Lemma 5.6. *The algorithm in Figure 5.3 satisfies the k-FCFS property.*

Proof. Let Y be a set of processes such that $|Y| = k$, and assume all processes in Y finish the doorway before a process p starts the doorway. Thus $\forall q \in Y$, $C_q \rightarrow C_p$. This and Lemma 5.1 (with the roles of p and q interchanged) imply that each process in Y chooses a ticket strictly smaller than the ticket that p chooses. By Lemma 5.3, during p's execution of the trying protocol, p does not remove any process in Y from $p.predecessor\_set$ until at least one process in Y completes the CS. This implies that the size of $p.predecessor\_set$ will be at least k until some process in Y executes through the CS. Process p does not enter the CS until the size of its predecessor set is less than k, and so p does not enter the CS until some process in Y enters the CS. □

Lemma 5.7. *Let p and q be distinct processes. In the CC model, while p is executing the loop at line 70, process p makes at most five RMRs reading $Want[q][p]$ at line 72.*

Proof. Suppose, by way of contradiction, that p makes six RMRs reading $Want[q][p]$ at line 72. Let $R_{\leq 5}$ be the operation in which p makes the first five of these RMRs to $Want[q][p]$, and let $R_{=5} \subseteq R_{\leq 5}$ be the operation in which p makes (only) its fifth RMR reading $Want[q][p]$. Recall that in the CC model a process makes an RMR when

it reads a variable for the first time, writes a variable, and whenever a process reads a variable for the first time after another process has written that variable. During $R_{\leq 5}$ (i.e., after $R_{\leq 5}$ starts and before $R_{\leq 5}$ finishes), q must update $Want[q][p]$ at least four times: p's first RMR in $R_{\leq 5}$ may happen because p does not have a copy of $Want[q][p]$ in its cache, but each of p's four subsequent RMRs to $Want[q][p]$ in $R_{\leq 5}$ can only occur after q writes $Want[q][p]$, invalidating the copy of $Want[q][p]$ in p's cache. This and the fact that process q writes $Want[q][p]$ exactly three times in a passage (line 66, line 68, and line 74) imply that there is a passage by q in which q executes lines 66..68 entirely during $R_{\leq 5}$ (i.e., q starts line 66 after $R_{\leq 5}$ starts and next finishes line 68 before $R_{\leq 5}$ finishes). In turn, this means that q chooses a ticket at line 67 (denote this operation C_q) during $R_{\leq 5}$. Let C_p be the operation in which p chose its ticket at line 67 prior to entering the loop at line 70. $C_p \rightarrow R_{\leq 5}$, and C_q happens during $R_{\leq 5}$, so $C_p \rightarrow C_q$. This and Lemma 5.1 imply that $t_p < t_q$, where t_p and t_q are the tickets chosen by p and q, respectively. After C_q finishes, q writes t_q to $Want[q][p]$ at line 68 (denote this operation A_q), which occurs before $R_{\leq 5}$ finishes, and in particular, before $R_{=5}$, p's fifth RMR to $Want[q][p]$. This and the fact that reads and writes are atomic, imply that $A_q \rightarrow R_{=5}$. If q subsequently writes to $Want[q][p]$ after A_q finishes, q always writes a value that is at least as large as t_q: at line 74, q writes ∞ to $Want[q][p]$, and in subsequent passages (if any), q announces at line 66 the ticket that it received in its previous passage, which is at least t_q, and announces at line 68 a ticket that is strictly larger than t_q. Thus, $Want[q][p] > t_p$ (i.e., $Want[q][p] > Ticket[p]$) is invariant from the start of $R_{=5}$ until p updates $Ticket[p]$ in its next passage (if any). This implies that after $R_{=5}$ (i.e., after p makes its fifth RMR to $Want[q][p]$ at line 72), p evaluates the condition at line 72 to be true and removes q from $p.predecessor\_set$. In turn, this implies that p does not make a sixth RMR to $Want[q][p]$ before entering the CS, contradicting the assumption that it does. □

Lemma 5.8. *A process makes $\Theta(N)$ RMRs in a passage of the algorithm in Figure 5.3*

in both the DSM and CC models.

Proof. We first do the proof for the CC model. A process p makes at most one RMR reading $Ticket[p]$ in the loop at line 70, since no process other than p writes $Ticket[p]$. This, Lemma 5.7, and the fact that there are N processes, imply that p makes at most $O(N)$ RMRs while p is executing the loop at line 70. Elsewhere in the algorithm, p makes $\Theta(N)$ RMRs, as follows. Process p announces itself twice to every other process in the system: once when it first leaves the NCS (line 66), and a second time after it chooses its ticket (line 68). Each of these announcements incurs $\Theta(N)$ RMRs. Choosing a ticket also incurs $\Theta(N)$ RMRs. Finally, in the exit protocol, p withdraws its announcement to enter the CS from every other process exactly once. This also incurs $\Theta(N)$ RMRs. Thus the RMR complexity is $\Theta(N)$ in the CC model.

The result also follows for the DSM model by a similar argument. The only difference in the DSM model is that p makes no RMRs while in the loop at line 70, as each variable $Ticket[p]$ and $Want[q][p]$ is local to p. □

Theorem 5.9. *The algorithm in Figure 5.3 satisfies k-exclusion, starvation freedom, and k-FCFS. Moreover, it has RMR complexity $\Theta(N)$ in both the DSM and CC models.*

Proof. The result follows from Lemmas 5.4, 5.5, 5.6, and 5.8. □

5.5 Space-optimal Local-spin k-Exclusion in the CC Model (Atomic RWs)

In Figure 5.4 we present a k-exclusion algorithm that uses only atomic reads and writes, has $O(N)$ RMR complexity in the CC model, and has $O(N)$ space complexity. Any mutual exclusion algorithm that uses only atomic reads and writes requires $\Omega(N)$ shared variables [9], and so this algorithm is space-optimal.[2]

[2]It is space-optimal in terms of the number of shared variables used; however, the shared variables can grow without bound.

The algorithm in Figure 5.4 is similar to the one in Figure 5.3, except that the *Want* array is one-dimensional. In particular, a process p writes its ticket into $Want[p]$ instead of writing its ticket into $Want[p][q]$ for each q separately. When p first leaves the NCS, it writes to $Want[p]$ the ticket that it chose in its previous passage (line 78). After this, p chooses a new ticket (line 79), and then writes the new ticket to $Want[p]$ (line 80). Each process q, when it reaches the loop at line 82, reads p's ticket from $Want[p]$. The correctness of the algorithm follows a nearly identical argument to the one made for the algorithm in Figure 5.3. The only place where an unbounded number of RMRs can potentially be made is in the loop at line 82. As a result of the features of the CC model, however, the RMR complexity of this algorithm is $O(N)$, which we prove below.

Figure 5.4: (CC/Atomic RWs) k-Exclusion algorithm for process $p \in \{1, ..., N\}$

```
   shared variables:
      Want: array[1..N] of ℕ ∪ {∞} init all ∞
      Ticket: array[1..N] of ℕ init all 0

   private variables:
      predecessor_set: Set of ℕ

76 loop
77 |  NCS
78 |  Want[p] := Ticket[p]
79 |  Ticket[p] := 1 + max(Ticket[1], Ticket[2], ..., Ticket[N])
80 |  Want[p] := Ticket[p]
81 |  predecessor_set := {1..N} \ {p}
82 |  while |predecessor_set| ≥ k do
83 |  |  foreach i ∈ predecessor_set do
84 |  |  |  if (Ticket[p], p) < (Want[i], i) then
   |  |  |  predecessor_set := predecessor_set \ {i}
85 |  CS
86 |  Want[p] := ∞
87 end loop
```

5.5.1 Proofs

Let C_p denote the operation in which process p chooses its ticket at line 79, and let EP_p denote the operation in which p executes its exit protocol at line 86. Using these definitions of C_p and EP_p, the statements of Lemma 5.1, Corollary 5.2, and Lemmas 5.3, 5.4, 5.5, 5.6 are true for the algorithm in Figure 5.4. The proofs are almost identical to the proofs given in Section 5.4, modulo line number changes, and so we do not reprove them here. It is understood that any time we reference these lemmas in this section, we are using them in the context of the algorithm in Figure 5.4.

The following lemma will allow us to prove that the algorithm has $O(N)$ RMR complexity in the CC model. The proof is nearly identical to the one for Lemma 5.7, modulo line number differences and the fact that the statement in Lemma 5.7 is about $Want[q][p]$ instead of $Want[q]$. As such, we state it without proof.

Lemma 5.10. *Let p and q be distinct processes. In the CC model, while p is executing the loop at line 82, process p makes at most five RMRs reading $Want[q]$ at line 84.*

Lemma 5.11. *A process makes at most $\Theta(N)$ RMRs in a passage of the algorithm given in Figure 5.4 in the CC model.*

Proof. A process makes at most 1 RMR to $Ticket[p]$ in the loop at line 82 since no process other than p writes $Ticket[p]$. This, Lemma 5.10, and the fact that there are N processes in the system, imply that a process makes $\Theta(N)$ RMRs in the loop at line 82. A process makes $\Theta(N)$ RMRs at line 79, $\Theta(1)$ RMRs at lines 78, 80, and 86, and no RMRs at line 81. Thus the total number of RMRs a process makes in a passage is $\Theta(N)$. □

Theorem 5.12. *The algorithm in Figure 5.4 satisfies k-exclusion, starvation freedom, and k-FCFS. Moreover, it has RMR complexity $\Theta(N)$ in the CC model.*

Proof. The result follows from Lemmas 5.4, 5.5, 5.6, and 5.11. □

5.6 FIFE k-Exclusion (Atomic RWs)

The algorithms presented in the preceding sections do not satisfy the FIFE property. To see why, we illustrate a scenario in which FIFE is violated by the algorithm in Figure 5.3. This scenario can be easily adapted for the algorithm in Figure 5.4.

Assume that $k = 2$, and consider the following execution of processes p_1, p_2, p_3, and p_4: Process p_1 executes its doorway entirely, choosing a ticket $t_{p_1} > 0$. After this, process p_2 executes its doorway and advances into the CS. Process p_3 and process p_4 then execute their doorway, choosing tickets t_{p_3} and t_{p_4}, both of which are larger than t_{p_1}. Process p_3 and p_4 then stop taking steps temporarily. We assume this is the first time processes p_3 and p_4 have left the NCS, and so at this point $Want[p_3][p_1] = 0$ and $Want[p_4][p_1] = 0$. This implies that when p_1 executes the loop at line 71, p_1 does not remove p_3 or p_4 from $p_1.predecessor\_set$, and hence does not advance into the CS, until either p_3 writes $Want[p_3][p_1] = t_{p_3}$ or p_4 writes $Want[p_4][p_1] = t_{p_4}$ (line 68). This violates FIFE, which requires that p_1 enters the CS in a bounded number of its own steps after p_2 enters the CS.

In this section we present a k-exclusion algorithm that satisfies FIFE, which turns out to be a simple modification of the algorithm presented in Figure 5.3.

The FIFE algorithm is presented in Figure 5.5, and the modifications to the original algorithm are shown shaded in gray. The doorway consists of lines 90..91.

FIFE says that if a process p finishes the doorway before a process q starts the doorway, and q enters the CS before p, then p enters the CS in a bounded number of its own steps. To ensure this, just before the process q enters the CS, q *captures* all processes that have a smaller ticket (line 99). In particular, since process p finishes the doorway before q starts the doorway, process p's ticket will be strictly smaller than q's ticket. Process q captures p by setting the value of $Capture[q][p]$ to q's current ticket. This and the fact that p has a smaller ticket than q, means that after q enters the CS, it will be

the case that $Ticket[p] < Capture[q][p]$. The next time process p evaluates the condition $Ticket[p] < Capture[q][p]$ at line 98, p sets $captured = \mathsf{true}$, allowing p to terminate the loop at line 95 and enter the CS in a bounded number of its own steps, as required to satisfy FIFE.

The addition of the capturing mechanism introduces a new way for processes to enter the CS. On first inspection, this calls into question whether the algorithm still satisfies k-exclusion, since it is not clear whether capturing can result in more than k processes in the CS. It turns out that this is not the case. Intuitively, the reason that this cannot happen is because a process only captures processes with smaller tickets. As a result, any process that is captured is "legitimately" allowed to advance into the CS. We provide a more precise proof of why k-exclusion holds in the next section.

5.6.1 Proofs

Let C_p denote the operation in which process p chooses its ticket at line 91, and let EP_p denote the operation in which p executes its exit protocol at line 101. Using these definitions of C_p and EP_p, the statements of Lemma 5.1, Corollary 5.2, Lemma 5.3, and Lemma 5.5 (starvation freedom) are true for the algorithm in Figure 5.5. The proofs are almost identical to the proofs given in Section 5.4, modulo line number changes, and so we do not reprove them here. It is understood that any time we reference these lemmas in this section, we are using them in the context of the algorithm in Figure 5.5.

The algorithm in this section introduces a capturing mechanism, which provides an additional path for a process to enter the CS. As a result, it is not immediately obvious that k-exclusion is satisfied. It is, and we prove this in Lemma 5.15. We also prove in Lemma 5.16 that FIFE is satisfied.

We start by defining the notion of capturing more precisely and by introducing two new lemmas (Lemma 5.13 and Lemma 5.14) that will help in the proof of k-exclusion.

Figure 5.5: (Atomic RWs) FIFE k-Exclusion algorithm for process $p \in \{1, ..., N\}$

shared variables:
Want: **array**$[1..N][1..N]$ **of** $\mathbb{N} \cup \{\infty\}$ **init all** ∞
Ticket: **array**$[1..N]$ **of** $\mathbb{N}$ **init all 0**
Capture: **array**$[1..N][1..N]$ **of** $\mathbb{N}$ **init all 0**
(DSM model: *Ticket*$[p]$, *Want*$[i][p]$, *Capture*$[i][p]$ are local to process p for all i)

private variables:
predecessor_set: **Set of** $\mathbb{N}$
captured: **boolean**

```
88  loop
89      NCS
90      foreach i ∈ {1..N} \ {p} do  Want[p][i] := Ticket[p]
91      Ticket[p] := 1 + max(Ticket[1], Ticket[2], ..., Ticket[N])
92      foreach i ∈ {1..N} \ {p} do  Want[p][i] := Ticket[p]
93      captured := false
94      predecessor_set := {1..N} \ {p}
95      while |predecessor_set| ≥ k ∧¬captured do
96          foreach i ∈ predecessor_set do
97              if (Ticket[p], p) < (Want[i][p], i) then
                predecessor_set := predecessor_set \ {i}
98          foreach i ∈ {1..N} do if Ticket[p] < Capture[i][p] then captured := true
99      foreach i ∈ {1..N} do Capture[p][i] := Ticket[p]
100     CS
101     foreach i ∈ {1..N} do  Want[p][i] := ∞
102 end loop
```

Let D_p^r be the operation in which a process p *detects capture* by a process r at line 98, i.e., D_p^r is the operation in which p reads *Ticket*$[p] <$ *Capture*$[r][p]$ and consequently sets $p.captured =$ true. Let X_r^p denote the operation in which a process r *attempts to capture* a process p at line 99, i.e., X_r^p is the operation in which r writes its ticket to *Capture*$[r][p]$ at line 99. The following lemma relates these two concepts by showing that when a process detects capture, there must exist some other process that previously attempted to capture it.

Lemma 5.13. *Suppose a process p detects capture by a process r in a passage in which*

it chooses a ticket t_p. Then there exists a passage by r in which the following are true: (i) r chooses a ticket $t_r > t_p$, and (ii) $X_r^p \dashrightarrow D_p^r$.

Proof. Process p detects capture by r in a passage in which it chooses ticket $t_p > 0$, and so it must evaluate the condition $t_p < \mathit{Capture}[r][p]$ to be true at line 98. Initially, $\mathit{Capture}[r][p] = 0$. This and the fact that reads and writes are atomic imply that some process must write a value larger than t_p before p reads $\mathit{Capture}[r][p]$. The only process that writes $\mathit{Capture}[r][p]$ is process r, at line 99, where it writes its ticket. Therefore there exists a passage by r in which it chooses a ticket $t_r > t_p$, and it writes that ticket to $\mathit{Capture}[r][p]$ before p reads it, meaning that $X_r^p \dashrightarrow D_p^r$. □

Let Q_p denote the first read operation in a passage that p executes after advancing past the loop at line 95. After Q_p finishes, we say that p is *CS-qualified*.

Lemma 5.14. *Let Y be a set of processes such that $|Y| \geq k$, and let p be a process that chooses a ticket t_p such that for all $i \in Y$, $(t_p, p) > (t_i, i)$, where t_i is the ticket chosen by i. Then p is not CS-qualified until some process in Y starts the exit protocol.*

Proof. Suppose, by way of contradiction, that p is CS-qualified before any process in Y starts the exit protocol, i.e., Q_p finishes before any process in Y starts the exit protocol. Further assume that the lemma is not violated prior to the start of Q_p. (If the lemma is violated earlier, then we can modify our choice of process p and its passage so that the preceding assumption holds.) By Lemma 5.3, p does not remove from $p.\mathit{predecessor\_set}$ any process in Y until some process in Y starts the exit protocol. This and the fact that $|Y| \geq k$ means that p cannot advance past the loop at line 95, and hence start Q_p, until either p sets $\mathit{captured} = \mathsf{true}$ or some process in Y starts the exit protocol. By assumption, Q_p finishes before any process in Y starts the exit protocol, and so p starts Q_p after p sets $\mathit{captured} = \mathsf{true}$, i.e., Q_p starts after p detects capture by some process r. This and Lemma 5.13 imply that there exists a passage by r in which the following are true: (i) r chooses a ticket $t_r > t_p$, and (ii) $X_r^p \dashrightarrow D_p^r$.

By (i), $t_r > t_p$, and by the fact that (t_p, p) is larger than any $(ticket, process\_id)$ pair chosen by a process in Y, it follows that r chooses a ticket t_r such that (t_r, r) is larger than any $(ticket, process\_id)$ pair in Y. By (ii), $Q_r \rightarrow X_r^p \dashrightarrow D_p^r \rightarrow Q_p$, and so $Q_r \rightarrow Q_p$. This and the assumption that Q_p finishes before any process in Y starts the exit protocol, imply that Q_r finishes before any process in Y starts the exit protocol, i.e., r is CS-qualified before any process in Y starts the exit protocol. Thus when r becomes CS-qualified the statement of the lemma is violated. By $Q_r \rightarrow Q_p$, r's passage violates the statement of the lemma prior to the start of Q_p, which contradicts the assumption that the lemma is not violated prior to the start of Q_p. □

Using the preceding lemma, it is straightforward to prove k-exclusion.

Lemma 5.15. *The algorithm in Figure 5.5 satisfies k-exclusion.*

Proof. Suppose, by way of contradiction, that $k+1$ processes are in the CS concurrently. Let Y be this set of processes, and let $p \in Y$ be the process in Y with the largest $(ticket, process\_id)$ pair. By Lemma 5.14, p is not CS-qualified until some process in $Y \setminus \{p\}$ leaves the CS, which means that p cannot be in the CS concurrently with all other processes in Y. This contradicts the assumption that p is in the CS concurrently with all other processes in Y. □

We now prove that FIFE holds.

Lemma 5.16. *The algorithm in Figure 5.5 satisfies FIFE.*

Proof. Let p and q be distinct processes that choose tickets t_p and t_q, respectively. Suppose that p finishes the doorway before q starts the doorway, and that q enters the CS before p. To prove that FIFE holds we show that after q enters the CS, p enters the CS in a bounded number of its own steps.

Process p finishes the doorway before q starts it, and therefore $C_p \rightarrow C_q$. This and Lemma 5.1 imply that $t_p < t_q$. Before q enters the CS, q writes t_q to $Capture[q][p]$ at

line 99. Moreover, the tickets chosen by q strictly increase with each passage that q executes, and so any time q subsequently writes to $Capture[q][p]$ it writes a value larger than t_q. This and the fact that $t_q > t_p$ imply that from the time that q enters the CS until p starts another passage $Capture[q][p] > t_p$ is invariant. This means that the first time that p evaluates the condition $t_p < Capture[q][p]$ at line 98 after q enters the CS, the condition evaluates to true, and p sets $captured$ = true. In turn, this and the fact that p executes each iteration of the loop in a bounded number of its own steps imply that after q enters the CS, p finishes the loop at line 95 in a bounded number of its own steps. Furthermore, by inspection, process p finishes line 99 in a bounded number of its own steps. Hence, after q enters the CS, p enters the CS in a bounded number of its own steps. □

The next lemma is used to prove that the algorithm has $\Theta(N)$ RMR complexity in the CC model. The proof is similar to the proof for Lemma 5.7, modulo line number differences, and so we state it without proof.

Lemma 5.17. *Let p and q be distinct processes. In the CC model, while p is executing the loop at line 95, process p makes at most five RMRs reading $Want[q][p]$ at line 97.*

Besides reading entries in the *Want* array in the loop at line 95, processes also read *Capture* entries. The following lemma is the analog to Lemma 5.17 for *Capture* entries.

Lemma 5.18. *Let p and q be distinct processes. In the CC model, while p is executing the loop at line 95, process p makes at most three RMRs reading $Capture[q][p]$ at line 98.*

Proof. Suppose, by way of contradiction, that p makes four RMRs reading $Capture[q][p]$ at line 98. Let $R_{\leq 3}$ be the operation in which p makes the first three of these RMRs to $Capture[q][p]$, and let $R_{=3} \subseteq R_{\leq 3}$ be the operation in which p makes (only) its third RMR reading $Capture[q][p]$. Recall that in the CC model a process makes an RMR when

it reads a variable for the first time, writes a variable, and whenever a process reads a variable for the first time after another process has written that variable. During $R_{\leq 3}$ (i.e., after $R_{\leq 3}$ starts and before $R_{\leq 3}$ finishes), q must update $Capture[q][p]$ at least two times: p's first RMR in $R_{\leq 3}$ may happen because p does not have a copy of $Capture[q][p]$ in its cache, but each of p's two subsequent RMRs to $Capture[q][p]$ in $R_{\leq 3}$ can only occur after q writes $Capture[q][p]$, invalidating the copy of $Capture[q][p]$ in p's cache. Process q writes $Capture[q][p]$ only once per passage at line 99, and so there must be two distinct passages in which q writes $Capture[q][p]$. Let W_0 be q's first write to $Capture[q][p]$, and W_1 be q's second write to $Capture[q][p]$ in its next passage. W_0 and W_1 both happen during $R_{\leq 3}$ (i.e., W_0 starts after $R_{\leq 3}$ starts and W_1 finishes before $R_{\leq 3}$ finishes). Let C_q be the operation in which q chooses its ticket at line 91 in the same passage as W_1. Let C_p be the operation in which p chose its ticket at line 91 prior to entering the loop at line 95.

$C_p \rightarrow R_{\leq 3}$, $W_0 \rightarrow C_q \rightarrow W_1$, and W_0 happens during $R_{\leq 3}$. Thus, $C_p \rightarrow C_q$. This and Lemma 5.1 imply that $t_p < t_q$, where t_p and t_q are the tickets chosen by p and q, respectively. Ticket t_q is the value that q writes to $Capture[q][p]$ in W_1. Furthermore, W_1 finishes before $R_{\leq 3}$ finishes, and in particular, before $R_{=3}$, p's third RMR reading $Capture[q][p]$, finishes. This and the fact that reads and writes are atomic, imply that $W_1 \rightarrow R_{=3}$. If q subsequently writes to $Capture[q][p]$ after W_1 finishes, q always writes a value that is at least as large as t_q: in subsequent passages (if any), q chooses a ticket strictly larger than t_q, and thus writes to $Capture[q][p]$ a value larger than t_q. Thus, $Capture[q][p] > t_p$ (i.e., $Capture[q][p] > Ticket[p]$) is invariant from the start of $R_{=3}$ until p updates $Ticket[p]$ in its next passage (if any). This implies that after $R_{=3}$ finishes (i.e., after p makes its third RMR to $Capture[q][p]$ at line 98), p evaluates the condition at line 98 to be true and sets $captured = \mathsf{true}$. In turn, this implies that p does not make a fourth RMR to $Capture[q][p]$ before entering the CS, contradicting the assumption that it does. □

Lemma 5.19. *A process makes $\Theta(N)$ RMRs in a passage of the algorithm in Figure 5.5 in both the DSM and CC models.*

Proof. We first do the proof for the CC model. A process p makes at most one RMR reading $Ticket[p]$ in the loop at line 95, since no process other than p writes $Ticket[p]$. This, Lemma 5.17, Lemma 5.18, and the fact that there are N processes in the system, imply that p makes $O(N)$ RMRs while in the loop at line 95. Elsewhere in the algorithm, p makes $\Theta(N)$ RMRs, as follows. Process p announces itself twice to every other process in the system: once when it first leaves the NCS (line 90), and a second time after it chooses its ticket (line 92). Each of these announcements incurs $\Theta(N)$ RMRs. Choosing a ticket also incurs $\Theta(N)$ RMRs. The capturing mechanism at line 99 incurs $\Theta(N)$ RMRs. Finally, in the exit protocol, p withdraws its announcement to enter the CS from every other process exactly once. This also incurs $\Theta(N)$ RMRs. Thus the RMR complexity is $\Theta(N)$ in the CC model.

The result also follows for the DSM model by a similar argument. The only difference in the DSM model is that p makes no RMRs while in the loop at line 95, as each variable $Ticket[p]$, $Want[q][p]$ and $Capture[q][p]$ is local to p. □

Theorem 5.20. *The algorithm in Figure 5.5 satisfies k-exclusion, starvation freedom, k-FCFS, and FIFE. Moreover, it has RMR complexity $\Theta(N)$ in both the DSM and CC models.*

Proof. Follows from Lemmas 5.15, 5.5, 5.16 and 5.19. □

5.7 FIFE k-Exclusion (Non-atomic RWs)

In this section we present a modification to the FIFE k-exclusion algorithm in Figure 5.5 that works when reads and writes are non-atomic. The algorithm is given in Figure 5.6. The trying protocol consists of lines 105..121, and the doorway consists of lines 105..107.

Firstly, we observe that none of the preceding algorithms satisfy k-exclusion, starvation freedom, or FIFE if reads and writes are non-atomic. The problem is that there are no guarantees on the value a process reads from a variable if some other process is concurrently writing the variable. Consider, for example, the algorithm in Figure 5.5, and the case where $k = 1$. Suppose processes p and q concurrently execute line 91, and that p and q choose tickets t_p and t_q, respectively. Further suppose that $t_q < t_p$ and that after p and q are both done choosing tickets (line 91), q temporarily stops taking steps. Process p races ahead of q into the loop at line 95, and starts reading $Want[q][p]$ at line 97. At the same time, q starts writing $Want[q][p]$ at line 92. Since the reads and writes are not atomic, p can read any value from $Want[q][p]$. In particular, suppose that p reads a value from $Want[q][p]$ such that $t_p < Want[q][p]$. In this case, p removes q from its predecessor set. All other processes are in the NCS, and so p removes them from its predecessor set as well and advances into the CS. When q reaches line 97, it evaluates $(t_q, q) < (Want[p][q], p)$ to be true, since the ticket t_p that p last wrote to $Want[p][q]$ is larger than the ticket t_q that q chose. Thus q will remove p from its predecessor set, along with all the other processes, which are in the NCS. After this, q advances into the CS, and k-exclusion is violated for $k = 1$. A slightly more complex scenario can be constructed for $k = 2$ to illustrate that starvation freedom and FIFE also do not hold.

To solve this problem, we employ a technique similar to one used by Lamport in his register constructions [34]. The technique involves reading variables in the opposite order of which they are written. We "duplicate" each $Want[i][j]$ and $Capture[i][j]$ variable in our algorithm. That is, for each i, j, we create $Want[i][j][1]$, $Want[i][j][2]$, $Capture[i][j][1]$, and $Capture[i][j][2]$. Wherever a write of value v to $Want[i][j]$ occurs in the algorithm in Figure 5.5, in the new algorithm we write v to $Want[i][j][1]$ and $Want[i][j][2]$, *in that order*. Wherever a read of variable $Want[i][j]$ occurs in the algorithm in Figure 5.5, in the new algorithm we read $Want[i][j][2]$ and $Want[i][j][1]$, *in that order*. Reading and writing $Capture[i][j][1]$ and $Capture[i][j][2]$ in the new algorithm is

done analogously. Note that in the algorithm we present the duplicate writes on the same line (e.g., line 106). We use this notation for compactness; the writes are still distinct (non-atomic) operations and occur in the order that they appear on the line.

There is also a new condition checked at line 120 that was not in the version of the algorithm with atomic reads and writes. In that version, there was no harm in p capturing i repeatedly in successive passages. Here, however, if p continues capturing i in each passage that p executes, it could be that every time that i checks to see if it is captured by p (on lines 117..118), it happens to read $Capture[p][i][1]$ and $Capture[p][i][2]$ as p is writing into them (on line 121) and so it does not find that it is captured. This neutralizes the capturing mechanism and can result in a violation of FIFE. To avoid the problem p does not attempt to capture any process it previously captured, by performing the test on line 120.

To gain some understanding about how the preceding changes fix the algorithm, consider again the problem scenario that we described above in which k-exclusion was violated. Process p and q leave the NCS for the first time, write 0 into all $Want$ entries in the loop at line 105, and then concurrently execute line 107. Process p chooses a ticket t_p larger than the ticket t_q that q chooses, i.e., $t_q < t_p$, after which process q temporarily stops taking steps. Process p then races ahead of q into the loop at line 112. When p tests the condition at line 114, the only way p evaluates it to be true is if q writes $Want[q][p][2]$ at the same time that p reads $Want[q][p][2]$. For q to write $Want[q][p][2]$ (line 109), it must first finish writing $Want[q][p][1]$ (line 109). Thus, when p checks the condition at line 115, p will evaluate it to be false unless q starts another write of $Want[q][p][1]$ concurrently with p's read of $Want[q][p][1]$. However, for this to happen, q must start another passage of the algorithm, in which case it is safe for p to remove q from p's predecessor set.

Figure 5.6: (Non-atomic RWs) FIFE k-Exclusion algorithm for process $p \in \{1, ..., N\}$

shared variables:
Want: **array**$[1..N][1..N][1..2]$ **of** $\mathbb{N} \cup \{\infty\}$ **init all** ∞
Ticket: **array**$[1..N]$ **of** $\mathbb{N}$ **init all 0**
Capture: **array**$[1..N][1..N][1..2]$ **of** $\mathbb{N}$ **init all 0**
(DSM model: $Ticket[p]$, $Want[i][p][j]$, $Capture[i][p][j]$ are local to process p for all i, j)

private variables:
predecessor_set: **Set of** $\mathbb{N}$
captured: **boolean**

```
103 loop
104   NCS
105   foreach i ∈ {1..N} \ {p} do
106     Want[p][i][1] := Ticket[p]; Want[p][i][2] := Ticket[p]
107   Ticket[p] := 1 + max(Ticket[1], Ticket[2], ..., Ticket[N])
108   foreach i ∈ {1..N} \ {p} do
109     Want[p][i][1] := Ticket[p]; Want[p][i][2] := Ticket[p]
110   predecessor_set := {1..N} \ {p}
111   captured := false
112   while |predecessor_set| ≥ k ∧ ¬captured do
113     foreach i ∈ predecessor_set do
114       if (Ticket[p], p) < (Want[i][p][2], i) then
115         if (Ticket[p], p) < (Want[i][p][1], i) then
            predecessor_set := predecessor_set \ {i}
116     foreach i ∈ {1..N} do
117       if Ticket[p] < Capture[i][p][2] then
118         if Ticket[p] < Capture[i][p][1] then captured := true
119   foreach i ∈ {1..N} do
120     if Capture[p][i][1] ≤ Ticket[i] then
121       Capture[p][i][1] := Ticket[p]; Capture[p][i][2] := Ticket[p]
122   CS
123   foreach i ∈ {1..N} do
124     Want[p][i][1] := ∞; Want[p][i][2] := ∞
125 end loop
```

5.7.1 Proofs

Let C_p denote the operation in which process p chooses its ticket at line 107, and let EP_p denote the operation in which p executes its exit protocol at lines 123..124.

The following lemma and its corollary are analogues to Lemma 5.1 and Corollary 5.2. The proof for the lemma below is nearly the same as Lemma 5.1, modulo line number differences.

Lemma 5.21. *Let p and q be distinct processes. If $C_p \rightarrow C_q$, then $t_p < t_q$, where t_p and t_q are the tickets chosen by p and q, respectively.*

Proof. Process p finishes line 107 before q starts it, and so the value that q reads from $Ticket[p]$ at line 107 is no smaller than t_p. This and inspection of line 107 imply that the ticket t_q that q chooses satisfies $t_q > t_p$. □

As a corollary to the preceding lemma, we state its contrapositive:

Corollary 5.22. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q. If $t_q \leq t_p$, then $C_q \dashrightarrow C_p$.*

The following lemma is an analogue to Lemma 5.3. The proof below is more complex than the one for Lemma 5.3, as it needs to take into account that entries in the *Want* array have been duplicated, and reads and writes are non-atomic.

Lemma 5.23. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q, respectively. If $(t_q, q) < (t_p, p)$, then p does not remove q from $p.predecessor\_set$ before EP_q starts.*

Proof. Suppose, by way of contradiction, that p removes q from $p.predecessor\_set$ at line 115 before EP_q starts. At line 115, p reads a value w_{q_1} in $Want[q][p][1]$ such that $(t_p, p) < (w_{q_1}, q)$ (denote this read operation R_1) before removing q from $p.predecessor\_set$ and therefore before EP_q starts. At line 114, before R_1 starts, p reads a value w_{q_2} in $Want[q][p][2]$ such that $(t_p, p) < (w_{q_2}, q)$ (denote this read operation R_2). Process q writes into $Want[q][p][1]$ and $Want[q][p][2]$ at most six times in its passage: at line 106 q writes the ticket $t'_q < t_q$ that q chose in its preceding passage (denote these writes W_{11} and

W_{12}); at line 109 q writes t_q (denote these writes W_{21} and W_{22}); and finally, at line 124, q writes ∞ (denote these writes W_{31} and W_{32}).

Claim 5.23.1. $R_2 \to R_1 \to W_{31} \to W_{32}$.

Proof. By the way we defined R_1 and R_2, $R_2 \to R_1$. Also, by the way we defined W_{31} and W_{32}, $W_{31} \to W_{32}$. It remains to show that $R_1 \to W_{31}$. This follows from the following two facts: (i) R_1 finishes before EP_q starts, and (ii) $W_{31} \subseteq EP_q$. (Claim 5.23.1) □

Claim 5.23.2. $W_{11} \to W_{12} \to R_2 \to R_1$.

Proof. By the way we defined W_{11} and W_{12}, $W_{11} \to W_{12}$. Also, by the way we defined R_1 and R_2, $R_2 \to R_1$. It remains to show that $W_{12} \to R_2$. By the assumption that $(t_q, q) < (t_p, p)$, $t_q \leq t_p$. This and Corollary 5.22 imply that $C_q \dashrightarrow C_p$. By inspection of the algorithm, $W_{12} \to C_q$ and $C_p \to R_2$. Thus, $W_{12} \to C_q \dashrightarrow C_p \to R_2$, and so $W_{12} \to R_2$. (Claim 5.23.2) □

There are two cases to consider: $R_2 \to W_{22}$, or $W_{22} \dashrightarrow R_2$. In the first case, by Claim 5.23.2, $W_{12} \to R_2 \to W_{22}$. W_{12} and W_{22} are successive writes into $Want[q][p][2]$, and so R_2 reads the value written by W_{12}, namely the ticket $t'_q < t_q$ that q chose in its preceding passage. By assumption, $(t_q, q) < (t_p, p)$, which implies that $t'_q < t_q \leq t_p$. That is, R_2 must read a value t'_q such that $t'_q < t_p$. This contradicts that R_2 reads a value w_{q_2} such that $(t_p, p) < (w_{q_2}, q)$, i.e., $w_{q_2} \geq t_p$.

In the second case, $W_{22} \dashrightarrow R_2$. This and definition of W_{21}, W_{22}, R_2, and R_1 imply that $W_{21} \to W_{22} \dashrightarrow R_2 \to R_1$. Therefore, $W_{21} \to R_1$. In turn, this and Claim 5.23.1 imply that $W_{21} \to R_1 \to W_{31}$. W_{21} and W_{31} are successive writes into $Want[q][p][1]$ and so the value w_{q_1} that R_1 reads must be the value t_q written by W_{21}, i.e., $w_{q_1} = t_q$. By assumption, $(t_q, q) < (t_p, p)$, which contradicts that $(t_p, p) < (w_{q_1}, q)$. □

For the atomic FIFE algorithm in Section 5.6, we defined some terminology regarding the capturing mechanism. We need to update this terminology for the non-atomic FIFE

algorithm in this section. Let D_q^p be the operation in which a process q *detects capture* by a process p at lines 117–118, i.e., D_q^p is the operation in which q reads $Ticket[q] < Capture[p][q][2]$ and $Ticket[q] < Capture[p][q][1]$ and consequently sets $q.captured = \mathsf{true}$ at line 118. Let X_p^q denote the operation in which a process p *attempts to capture* a process q at line 121, i.e., X_p^q is the operation in which p writes its ticket to $Capture[q][p][1]$ and $Capture[q][p][2]$ at line 121. Note that these operations are non-atomic.

The following lemma relates capture detection and capture attempts by showing that when a process detects capture, there must exist some other process that previously attempted to capture it. The lemma is an analogue to Lemma 5.13, but its proof is more complex since it needs to take into account that entries in the *Capture* array are duplicated, and the fact that reads and writes are non-atomic.

Lemma 5.24. *Suppose a process p detects capture by a process r in a passage in which p chooses a ticket t_p. Then there exists a passage by process r in which the following are true: (i) r chooses a ticket $t_r > t_p$, and (ii) $X_r^p \dashrightarrow D_p^r$.*

Proof. Process p detects capture by a process r in a passage in which it chooses a ticket $t_p > 0$. This implies that p reads a value w_{r1} from $Capture[r][p][1]$ at line 118 such that $t_p < w_{r1}$ (denote this read operation R_1), and prior to this reads a value w_{r2} from $Capture[r][p][2]$ at line 117 such that $t_p < w_{r2}$ (denote this read operation R_2). Initially, $Capture[r][p][1] = 0$ and $Capture[r][p][2] = 0$, and so some process must start writing to $Capture[r][p][1]$ and $Capture[r][p][2]$ before D_p^r ends. The only process that writes $Capture[r][p][1]$ and $Capture[r][p][2]$ is process r, at line 121, and the only value that r writes to $Capture[r][p][1]$ and $Capture[r][p][2]$ is the ticket that r chose in its current passage. This means that there exists a passage by r in which r starts an attempt to capture p before D_p^r ends, i.e., $X_r^p \dashrightarrow D_p^r$. Consider the last such passage by r prior to the end of D_p^r. It remains to show that in this passage r chooses a ticket $t_r > t_p$. Suppose, by way of contradiction, that r chooses ticket $t_r \leq t_p$ in this passage. Let W_1 be the write by r in which it writes t_r to $Capture[r][p][1]$ at line 121, and let W_2 be the write by

r in which it writes t_r to $Capture[r][p][2]$ at line 121. Either $W_1 \rightarrow R_1$ or $R_1 \dashrightarrow W_1$:

Case 1: $W_1 \rightarrow R_1$. By definition, $R_1 \subseteq D_p^r$, and W_1 is the last write to $Capture[r][p][1]$ that starts prior to the end of D_p^r. This and $W_1 \rightarrow R_1$ imply that W_1 is the last write to $Capture[r][p][1]$ before R_1 starts. Therefore, R_1 reads the value written by W_1. That is, R_1 reads t_r, and so $w_{r1} = t_r$. By assumption, however, $t_r \leq t_p$, which contradicts that $w_{r1} > t_p$.

Case 2: $R_1 \dashrightarrow W_1$. We first argue that $R_2 \rightarrow W_2$, and then show that this gives rise to a contradiction.

Claim 5.24.1. $R_2 \rightarrow W_2$

Proof. We have that $R_2 \rightarrow R_1 \dashrightarrow W_1 \rightarrow W_2$, and so $R_2 \rightarrow W_2$. (Claim 5.24.1) □

Let C_p denote the operation in which p chooses ticket t_p at line 107, and let C_r denote the operation in which r chooses ticket t_r at line 107. Let W_2' be the last write by r to $Capture[r][p][2]$ at line 121 before W_2. (W_2' must exist, otherwise $R_2 \rightarrow W_2$ implies that R_2 reads the initial 0 value from $Capture[r][p][2]$, contradicting that R_2 reads a value larger than t_p, where $t_p > 0$.) Let t_r' be the ticket written by W_2'. Either $R_2 \dashrightarrow W_2'$ or $W_2' \rightarrow R_2$:

Case 2a: $R_2 \dashrightarrow W_2'$. In this case, $C_p \rightarrow R_2 \dashrightarrow W_2' \rightarrow C_r$, and so $C_p \rightarrow C_r$. By Lemma 5.21, $t_p < t_r$, which contradicts the assumption that $t_r \leq t_p$.

Case 2b: $W_2' \rightarrow R_2$. By Claim 5.24.1, $W_2' \rightarrow R_2 \rightarrow W_2$. W_2' and W_2 are successive writes to $Capture[r][p][2]$ and so R_2 reads t_r', the ticket written by r in W_2'. The ticket t_r' must be smaller than t_r, since tickets strictly increase with each passage that a process executes, and W_2' occurs in a passage that precedes the one in which r chooses t_r. This and the assumption that $t_r \leq t_p$ imply that the value t_r' returned by R_2 is smaller than t_p, which contradicts the fact that the value $w_{r2} = t_r'$ returned by R_2 is greater than t_p. □

Let Q_p denote the first read operation in a passage that p executes after advancing past the loop at line 112. After Q_p finishes, we say that p is *CS-qualified*.

The following lemma is an analogue to Lemma 5.14, and is critical for proving k-exclusion. Its proof turns out to be very similar to the proof of Lemma 5.14. The reason for this similarity is because the proof relies on Lemma 5.24, and the proof of Lemma 5.14 relies on the analogue of Lemma 5.24, i.e., Lemma 5.13. The introduction of non-atomic reads and writes in the algorithm of this section only affects the proof of Lemma 5.24, but not (at least not directly) the proof below.

Lemma 5.25. *Let Y be a set of processes such that $|Y| \geq k$, and let p be a process that chooses a ticket t_p such that for all $i \in Y$, $(t_p, p) > (t_i, i)$, where t_i is the ticket chosen by i. Then p is not CS-qualified until some process in Y starts the exit protocol.*

Proof. Suppose, by way of contradiction, that p is CS-qualified before any process in Y starts the exit protocol, i.e., Q_p finishes before any process in Y starts the exit protocol. Further assume that the lemma is not violated prior to the start of Q_p. (If the lemma is violated earlier, then we can modify our choice of process p and its passage so that the preceding assumption holds.) By Lemma 5.23, p does not remove from *p.predecessor_set* any process in Y until some process in Y starts the exit protocol. This and the fact that $|Y| \geq k$ means that p cannot advance past the loop at line 112, and hence start Q_p, until either p sets *captured* = true or some process in Y starts the exit protocol. By assumption, Q_p finishes before any process in Y starts the exit protocol, and so p starts Q_p after p sets *captured* = true, i.e., Q_p starts after p detects capture by some process r. This and Lemma 5.24 imply that there exists a passage by r in which the following are true: (i) r chooses a ticket $t_r > t_p$, and (ii) $X_r^p \dashrightarrow D_p^r$.

By (i), $t_r > t_p$, and by the assumption that for all $i \in Y$, $(t_p, p) > (t_i, i)$, it follows that r chooses a ticket t_r such that for all $i \in Y$, $(t_r, r) > (t_i, i)$. By (ii), $Q_r \rightarrow X_r^p \dashrightarrow D_p^r \rightarrow Q_p$, and so $Q_r \rightarrow Q_p$. This and the assumption that Q_p finishes before any process in Y starts the exit protocol, imply that Q_r finishes before any process in Y starts the exit protocol, i.e., r is CS-qualified before any process in Y starts the exit protocol. Thus when r becomes CS-qualified the statement of the lemma is violated. By $Q_r \rightarrow Q_p$, r's

passage violates the statement of the lemma prior to the start of Q_p, which contradicts the assumption that the lemma is not violated prior to the start of Q_p. □

Using the preceding lemma, it is straightforward to prove k-exclusion.

Lemma 5.26. *The algorithm in Figure 5.6 satisfies k-exclusion.*

Proof. Suppose, by way of contradiction, that $k + 1$ processes are in the CS concurrently. Let Y be this set of processes, and let $p \in Y$ be the process with the largest $(ticket, process\_id)$ pair. By Lemma 5.25, p is not CS-qualified until some process in $Y \setminus \{p\}$ leaves the CS, which means that p cannot be in the CS concurrently with all other processes in Y. This contradicts the assumption that p is in the CS concurrently with all other processes in Y. □

The following lemma is used to prove that FIFE holds, and that a process has $O(N)$ RMR complexity in the CC model.

Lemma 5.27. *If a process p is non-faulty and finishes the doorway before a process q starts the doorway, then after q enters the CS, p enters the CS after taking $O(N)$ steps.*

Proof. Assume that a process p is non-faulty and finishes the doorway before a process q starts the doorway. We need to show that after q enters the CS, p enters the CS after taking $O(N)$ steps. Suppose, for contradiction, that this is not the case. Let CS_p be the operation in which p executes the CS, and let CS_q be the operation in which q executes its first step in the CS. (If CS_q does not exist, i.e., q never enters the CS, then the lemma holds, so assume that CS_q exists. We have no guarantee at this point that p enters the CS, but we define the CS_p operation now and are careful to make sure it exists if necessary in our reasoning below.) Furthermore, let t_p and t_q be the tickets chosen by p and q, respectively, at line 107.

Claim 5.27.1. $t_p < t_q$.

Proof. Since p finishes the doorway before q starts the doorway, $C_p \to C_q$. Hence, by Lemma 5.21, $t_p < t_q$. □

Let R_q^T be the read by process q of $Ticket[p]$ at line 120. (This read exists by the structure of the algorithm and the fact that CS_q exists.) Let $W_p^T \subseteq C_p$ be p's write of t_p to $Ticket[p]$ at line 107. (This operation exists since p finishes the doorway by assumption).

Claim 5.27.2. *If $\hat{R}$ is R_q^T, or $\hat{R}$ is a read by q of $Ticket[p]$ at line 120 such that $R_q^T \to \hat{R} \to CS_p$, then $\hat{R}$ returns the same value as written by W_p^T, i.e. t_p.*

Proof. Assume that $\hat{R}$ is R_q^T, or that $\hat{R}$ is a read by q of $Ticket[p]$ at line 120 such that $R_q^T \to \hat{R} \to CS_p$. Also, let W_p^{T+} be p's first write to $Ticket[p]$ after W_p^T, i.e., W_p^{T+} is p's write to $Ticket[p]$ at line 107 in its next passage. (Note that W_p^{T+} may not exist.)

The assumption that process p finishes the doorway before q starts the doorway, and the fact that W_p^T and C_q are part of the doorway, imply $W_p^T \to C_q \to R_q^T$, and so $W_p^T \to R_q^T$. Therefore $W_p^T \to \hat{R}$ (since $\hat{R}$ is R_q^T or $R_q^T \to \hat{R}$). Process p is the only process that writes $Ticket[p]$, and so if W_p^{T+} does not exist, then W_p^T is the uniquely defined last write to $Ticket[p]$ that finishes before $\hat{R}$ starts, and we're done. So assume that W_p^{T+} exists. We will now show that $\hat{R} \to W_p^{T+}$. Once we do this, since p is the only process to write $Ticket[p]$, and W_p^T and W_p^{T+} are successive writes by p, it follows that $\hat{R}$ must read the value written by W_p^T, i.e., t_p. It remains to show that $\hat{R} \to W_p^{T+}$.

Suppose, for contradiction, that $W_p^{T+} \dashrightarrow \hat{R}$. Since W_p^{T+} exists and is a write at line 107 in p's next passage, it follows that CS_p exists. Furthermore, by the structure of the algorithm and the assumption that $\hat{R} \to CS_p$, we have that $CS_p \to W_p^{T+} \dashrightarrow \hat{R} \to CS_p$, which implies that $CS_p \to CS_p$. This contradicts that $\to$ is an irreflexive relation. Thus $\hat{R} \to W_p^{T+}$, as desired. □

Let R_{q1}^C be q's read of $Capture[q][p][1]$ at line 120. (This reads exists by the structure of the algorithm and the fact that CS_q exists.) Furthermore, let R_{p2}^C and R_{p2}^{C+} be p's first

and second (in the sense of the total order given by $\rightarrow$) operations in the same passage by p that read $Capture[q][p][2]$ at line 117 such that $CS_q \dashrightarrow R^C_{p2}$. (These operations exist by inspection of the algorithm, the assumption that p is non-faulty, and the assumption that after q enters the CS, p does not enter the CS after taking $O(N)$ steps.)

Either q evaluates the condition at line 120 to be true or not.

Case 1: q evaluates the condition at line 120 to be true.

In this case, let W^C_{q1} and W^C_{q2} be q's write of t_q to $Capture[q][p][1]$ and $Capture[q][p][2]$ at line 121. (These writes exist by the structure of the algorithm, the assumption of this case, and the assumption that CS_q exists).

Recall that R^C_{p2} and R^{C+}_{p2} are p's first and second read of $Capture[q][p][2]$ at line 117 such that $CS_q \dashrightarrow R^C_{p2}$.

Claim 5.27.3. *R^{C+}_{p2} reads t_q.*

Proof. By $CS_q \dashrightarrow R^C_{p2}$ and the structure of the algorithm, $W^C_{q2} \rightarrow CS_q \dashrightarrow R^C_{p2} \rightarrow R^{C+}_{p2}$, and so $W^C_{q2} \rightarrow R^{C+}_{p2}$. We will now show that R^{C+}_{p2} reads the same value as written by W^C_{q2}, i.e., t_q. To do this, we show that for any write $\hat{W}$ to $Capture[q][p][2]$ other than W^C_{q2}, either $\hat{W} \rightarrow W^C_{q2}$ or $R^{C+}_{p2} \rightarrow \hat{W}$.

Process q is the only process that writes $Capture[q][p][2]$. For any write $\hat{W}$ to $Capture[q][p][2]$ by q other than W^C_{q2}, either $\hat{W} \rightarrow W^C_{q2}$ or $W^C_{q2} \rightarrow \hat{W}$. If $\hat{W} \rightarrow W^C_{q2}$, then we're done. So assume that $W^C_{q2} \rightarrow \hat{W}$. It suffices to show that $R^{C+}_{p2} \rightarrow \hat{W}$ for the first write $\hat{W}$ by q such that $W^C_{q2} \rightarrow \hat{W}$. Suppose, for contradiction that $\hat{W} \dashrightarrow R^{C+}_{p2}$.

Let $R^{C'}_{q1}$ be q's read of $Capture[q][p][1]$ at line 120 immediately before $\hat{W}$, and let $R^{T'}_q$ be q's read of $Ticket[p]$ at line 120 immediately before $\hat{W}$. (These operations exist by the structure of the algorithm.) The condition at line 120 must be true for $\hat{W}$ to exist, and so $R^{C'}_{q1}$ returns a value that is $\leq$ the value returned by $R^{T'}_q$.

By the assumption that $\hat{W} \dashrightarrow R^{C+}_{p2}$, and the structure of the algorithm, we have that $R^{T'}_q \rightarrow \hat{W} \dashrightarrow R^{C+}_{p2} \rightarrow CS_p$, and so $R^{T'}_q \rightarrow CS_p$. Furthermore, $R^T_q \rightarrow R^{T'}_q$, and so

$R_q^T \rightarrow R_q^{T'} \rightarrow CS_p$. By Claim 5.27.2, $R_q^{T'}$ returns the same value as written by W_p^T, i.e., t_p. We will now prove that $R_{q1}^{C'}$ returns t_q. Once we do so, we can make the following argument. In the preceding paragraph we established that $R_{q1}^{C'}$ returns a value that is $\leq$ the value returned by $R_q^{T'}$, and so $t_q \leq t_p$, i.e. $t_p \geq t_q$. This contradicts Claim 5.27.1, which says that $t_p < t_q$. (Thus, the supposition that $\hat{W} \dashrightarrow R_{p2}^{C+}$ will be false, and $R_{p2}^{C+} \rightarrow \hat{W}$, as desired.) It remains to prove that $R_{q1}^{C'}$ returns t_q.

$R_{q1}^{C'}$ occurs in later passage by q than W_{q1}^C, and so $W_{q1}^C \rightarrow R_{q1}^{C'}$. We will prove that $R_{q1}^{C'}$ returns the same value written as W_{q1}^C, i.e., t_q. To do this, we show that for any write $\bar{W}$ to *Capture*$[q][p][1]$ other than W_{q1}^C, either $\bar{W} \rightarrow W_{q1}^C$ or $R_{q1}^{C'} \rightarrow \bar{W}$.

Process q is the only process that writes *Capture*$[q][p][1]$. Recall that $\hat{W}$ is the first write by q to *Capture*$[q][p][2]$ after W_{q2}^C, and $R_{q1}^{C'}$ is the read of *Capture*$[q][p][1]$ that immediately precedes $\hat{W}$. Process q writes *Capture*$[q][p][1]$ and *Capture*$[q][p][2]$ at most once per passage (at line 121), after the condition at line 120 is evaluated. So, by the structure of the algorithm, the first write by q to *Capture*$[q][p][1]$ after W_{q1}^C finishes can start only after $R_{q1}^{C'}$ finishes. Therefore, for any write $\bar{W}$ by q to *Capture*$[q][p][1]$ other than W_{q1}^C, either $\bar{W} \rightarrow W_{q1}^C$ (if $\bar{W}$ occurs in a passage before the one in which W_{q1}^C and W_{q2}^C occur), or $R_{q1}^{C'} \rightarrow \bar{W}$ (if $\bar{W}$ occurs after W_{q1}^C finishes), as desired. □

By Claim 5.27.3, R_{p2}^{C+} reads t_q, and by Claim 5.27.1, $t_p < t_q$. Thus p evaluates the condition at line 117 to be true. An argument similar to the one in Claim 5.27.3 can be used to establish that the value returned by p's next read of *Capture*$[q][p][1]$ at line 118 is t_q. Using this, we can establish that the condition p evaluates at line 118 is true. After this point, p sets *captured* to be `true`, and, by inspection of the algorithm, executes $O(N)$ more steps prior to entering the CS.

We have established the following: After q enters the CS, p executes $O(N)$ steps before starting R_{p2}^{C+} at line 117. This follows by the structure of the algorithm and because R_{p2}^C and R_{p2}^{C+} are p's first and second reads of *Capture*$[q][p][2]$ at line 117 such that $CS_q \dashrightarrow R_{p2}^C$. After finishing R_{p2}^{C+}, p executes $O(N)$ more steps prior to entering

the CS. Therefore, after q enters the CS, p executes $O(N)$ steps in the trying protocol before entering the CS. This contradicts that after q enters the CS, p does not enter the CS after taking $O(N)$ steps.

Case 2: q evaluates the condition at line 120 to be false.

By Claim 5.27.2, R_q^T returns t_p. By assumption of this case, q evaluates the condition at line 120 to be false, and so R_{q1}^C returns a value strictly greater than t_p.

Recall that R_{p2}^C and R_{p2}^{C+} are p's first and second read of $Capture[q][p][2]$ at line 117 such that $CS_q \dashrightarrow R_{p2}^C$.

Claim 5.27.4. *R_{p2}^{C+} reads the same value as R_{q1}^C.*

Proof. By the structure of the algorithm and the assumption that $CS_q \dashrightarrow R_{p2}^C$, $R_{q1}^C \to CS_q \dashrightarrow R_{p2}^C \to R_{p2}^{C+}$, and so $R_{q1}^C \to R_{p2}^{C+}$. Process q is the only process that writes $Capture[q][p][1]$ and $Capture[q][p][2]$. We show that for any write $\hat{W}$ by q to $Capture[q][p][1]$ (resp. $Capture[q][p][2]$), either $\hat{W} \to R_{q1}^C$ (i.e., $\hat{W}$ occurs in a passage preceding the one in which R_{q1}^C occurs), or $R_{p2}^{C+} \to \hat{W}$. Once we do so, this, and the fact that process q writes the same value to $Capture[q][p][1]$ and $Capture[q][p][2]$ at most once per passage at line 121, imply that R_{p2}^{C+} reads the same value as R_{q1}^C. It remains to show that for any write $\hat{W}$ by q to $Capture[q][p][1]$ (resp. $Capture[q][p][2]$), either $\hat{W} \to R_{q1}^C$, or $R_{p2}^{C+} \to \hat{W}$.

By assumption of this case, q does not write $Capture[q][p][1]$ (resp. $Capture[q][p][2]$) in the same passage as R_{q1}^C occurs. Thus, for any write $\hat{W}$ to $Capture[q][p][1]$ (resp. $Capture[q][p][2]$) by q, either $\hat{W} \to R_{q1}^C$ or $R_{q1}^C \to \hat{W}$. If $\hat{W} \to R_{q1}^C$, then we are done. So assume that $R_{q1}^C \to \hat{W}$. It suffices to show that $R_{p2}^{C+} \to \hat{W}$ for the first write $\hat{W}$ to $Capture[q][p][1]$ (resp. $Capture[q][p][2]$) by q such that $R_{q1}^C \to \hat{W}$. Suppose, for contradiction that $\hat{W} \dashrightarrow R_{p2}^{C+}$.

Let $R_{q1}^{C'}$ be q's read of $Capture[q][p][1]$ at line 120 immediately before $\hat{W}$, and let $R_q^{T'}$ be q's read of $Ticket[p]$ at line 120 immediately before $\hat{W}$. (These operations exist by

the structure of the algorithm.) The condition at line 120 must be true for $\hat{W}$ to exist, so $R_{q1}^{C'}$ returns a value that is $\leq$ the value returned by $R_q^{T'}$.

By the assumption that $\hat{W} \dashrightarrow R_{p2}^{C+}$, and the structure of the algorithm, we have that $R_q^{T'} \rightarrow \hat{W} \dashrightarrow R_{p2}^{C+} \rightarrow CS_p$, and so $R_q^{T'} \rightarrow CS_p$. This, and the fact that $R_q^{T'}$ occurs in a later passage than R_q^T, imply $R_q^T \rightarrow R_q^{T'} \rightarrow CS_p$. So, by Claim 5.27.2, $R_q^{T'}$ returns the same value as written by W_p^T, i.e., t_p. In the preceding paragraph we established that $R_{q1}^{C'}$ returns a value v that is $\leq$ the value returned by $R_q^{T'}$, and so $v \leq t_p$. We will now prove that $R_{q1}^{C'}$ returns the same value v as R_{q1}^C. Once we do so, we can make the following argument. Recall that by assumption of this case, the value returned by R_{q1}^C is strictly greater than t_p. Therefore $v > t_p$, which contradicts that $v \leq t_p$. Thus, the supposition that $\hat{W} \dashrightarrow R_{p2}^{C+}$ is false, and $R_{p2}^{C+} \rightarrow \hat{W}$, as desired. It remains to prove that $R_{q1}^{C'}$ returns the same value as R_{q1}^C.

$R_{q1}^{C'}$ occurs in a passage after R_{q1}^C, so $R_{q1}^C \rightarrow R_{q1}^{C'}$. To show that $R_{q1}^{C'}$ returns the same value as R_{q1}^C, we show that for any write $\bar{W}$ to $Capture[q][p][1]$, $\bar{W} \rightarrow R_{q1}^C$ or $R_{q1}^{C'} \rightarrow \bar{W}$.

Process q is the only processes that writes $Capture[q][p][1]$. Recall that $\hat{W}$ is the first write by q to $Capture[q][p][1]$ (resp. $Capture[q][p][2]$) after R_{q1}^C, and $R_{q1}^{C'}$ is the read of $Capture[q][p][1]$ by q that immediately precedes $\hat{W}$. Process q writes $Capture[q][p][1]$ and $Capture[q][p][2]$ at most once per passage (at line 121), after the condition at line 120 is evaluated, and no writes by q to $Capture[q][p][1]$ or $Capture[q][p][2]$ occur in the same passage that R_{q1}^C occurs. Therefore, for any write $\bar{W}$ by q to $Capture[q][p][1]$, either $\bar{W} \rightarrow R_{q1}^C$ (if $\bar{W}$ occurs in a passage before the one in which R_{q1}^C occurs), or $R_{q1}^{C'} \rightarrow \bar{W}$ (if $\bar{W}$ occurs in the same or later passage as $\hat{W}$), as desired. □

By Claim 5.27.4, R_{p2}^{C+} returns the same value as R_{q1}^C. Immediately before Claim 5.27.4, we established that R_{q1}^C returns a value strictly greater than t_p. Thus p evaluates the condition at line 117 to be true. An argument similar to the one in Claim 5.27.4 can be used to establish that the value returned by p's next read of $Capture[q][p][1]$ at line 118 is the same as the value returned by R_{q1}^C. Using this, we can establish that the condition

p evaluates at line 118 is true. After this point, p sets *captured* to be true, and, by inspection of the algorithm, executes $O(N)$ more steps prior to entering the CS.

We have established the following: After q enters the CS, p executes $O(N)$ steps before starting R_{p2}^{C+} at line 117. This follows by the structure of the algorithm and because R_{p2}^{C} and R_{p2}^{C+} are p's first and second reads of $Capture[q][p][2]$ at line 117 such that $CS_q \dashrightarrow R_{p2}^{C}$. After finishing R_{p2}^{C+}, p executes $O(N)$ more steps prior to entering the CS. Therefore, after q enters the CS, p executes $O(N)$ steps in the trying protocol before entering the CS. This contradicts that after q enters the CS, p does not enter the CS after taking $O(N)$ steps. □

Lemma 5.28. *The algorithm in Figure 5.6 satisfies FIFE.*

Proof. This result follows immediately from Lemma 5.27. □

We next prove that deadlock freedom holds. We then prove that the FIFE property (Lemma 5.28) together with deadlock freedom imply starvation freedom.

Recall the alternate statement of the deadlock freedom property for k-exclusion from Section 1.4:

Deadlock Freedom: If a non-faulty process p is stuck in the trying protocol forever, and at most $k-1$ processes crash, then some process executes an infinite number of passages through the CS.

Lemma 5.29. *The algorithm in Figure 5.6 satisfies deadlock freedom.*

Proof. Suppose, by way of contradiction, that the algorithm is not deadlock free. Thus there is a set $Y \neq \emptyset$ of live processes so that eventually all processes in Y are stuck forever in the trying protocol and all other processes are forever in the NCS or crash. Let p be the process in Y with the smallest $(ticket, process\_id)$ pair, and let t_p be the ticket that p chooses. All the processes in Y eventually reach the loop at line 112, and so by inspection of the algorithm, eventually no shared variables are ever written after a

certain point in the execution. For each process q that is eventually in the NCS forever, the last value that q writes to $Want[q][p][1]$ and $Want[q][p][2]$ is ∞. This and inspection of lines 114..115 imply that p eventually removes from its predecessor set all processes that are eventually in the NCS forever. For each process $q \in Y \setminus \{p\}$, the last value t_q that q writes to $Want[q][p][1]$ and $Want[q][p][2]$ is such that $(t_p, p) < (t_q, q)$. This and inspection of lines 114..115 imply that p eventually removes from its predecessor set all processes in $Y \setminus \{p\}$. Therefore the only processes that eventually remain in p's predecessor set are faulty processes. There are at most $k-1$ faulty processes, and so p eventually finishes the loop at line 112 and enters the CS. This contradicts that p is stuck in the trying protocol forever. □

Lemma 5.30. *The algorithm in Figure 5.6 satisfies starvation freedom.*

Proof. Suppose, by way of contradiction, that some non-faulty process p is stuck in the trying protocol forever, and at most $k-1$ other processes have crashed. By Lemma 5.29, some process q executes through the CS infinitely often. Moreover, the only place where p can get stuck forever is in the loop at line 112. The second time that q executes through the CS while p is stuck in the loop at line 112, q will have started the doorway after p already finished it. Thus, by the FIFE property (Lemma 5.28), after q enters the CS, p can enter the CS in a bounded number of its own steps. This contradicts that p is stuck in the trying protocol forever. □

Lemma 5.31. *In the CC model, a process p makes $O(N)$ RMRs in the loop at line 112.*

Proof. Either process p makes at most $40N$ RMRs in the loop at line 112, or at least $40N+1$ RMRs. In the former case, the lemma is clearly true. In the latter case, let $R_{\leq 40}$ denote the operation in which p makes its first $40N+1$ RMRs. Process p makes at most one RMR in $R_{\leq 40}$ to the variable $Ticket[p]$, since no process other than p ever writes that variable. Besides $Ticket[p]$, there are at most $4N$ other shared variables ($Want[i][p][1]$,

$Want[i][p][2]$, $Capture[i][p][1]$, and $Capture[i][p][2]$, for each $i \in \{1..N\}$) that p reads in the loop, and so p makes at least ten RMRs to one of these variables before $R_{\leq 40}$ finishes. We consider the possible variables in four separate cases:

Case 1: Process p makes at least ten RMRs to $Want[q][p][2]$ for some process q. Let $R_{\leq 10} \subseteq R_{\leq 40}$ be the operation in which p makes these RMRs, and let $R_p^i \subseteq R_{\leq 10}$ for $i \in \{1..10\}$ be the operation in which p makes (only) its i'th RMR in $R_{\leq 10}$. Recall that in the CC model a process makes an RMR when it reads a variable for the first time, writes a variable, and whenever a process reads a variable for the first time after another process has written that variable. During $R_{\leq 10}$ (i.e., after $R_{\leq 10}$ starts and before $R_{\leq 10}$ finishes), q must update $Want[q][p][2]$ at least nine times: p's first RMR in $R_{\leq 10}$ may happen because p does not have copy of $Want[q][p][2]$ in its cache, but each of p's nine subsequent RMRs to $Want[q][p][2]$ in $R_{\leq 10}$ can only occur after q writes $Want[q][p][2]$, invalidating the copy of $Want[q][p][2]$ in p's cache. More precisely, there must exist distinct write operations W_q^j by q to $Want[q][p][2]$, for $j \in \{1..9\}$, such that $R_p^j \dashrightarrow W_q^j \rightarrow R_p^{j+1}$. This and the fact that process q writes $Want[q][p][2]$ exactly three times in a passage (line 106, line 109, and line 124) imply that there is a passage by q that occurs during $R_{\leq 10}$, (i.e., q's passage starts after $R_{\leq 10}$ starts and finishes before $R_{\leq 10}$ finishes). More precisely, there is a passage P_q by process q such that $R_p^1 \rightarrow P_q$ and $P_q \rightarrow R_p^{10}$. Let the operation DWY_q denote q's execution of its doorway in P_q, and let DWY_p denote p's execution of its doorway. $DWY_p \rightarrow R_p^1 \rightarrow P_q$, and so $DWY_p \rightarrow R_p^1 \rightarrow DWY_q$. This implies that p finishes its doorway before q starts its doorway. By this and Lemma 5.27, after q enters the CS, p enters the CS after taking $O(N)$ steps. Process q's passage P_q finishes before R_p^{10} starts (since $P_q \rightarrow R_p^{10}$), and so q enters the CS before p finishes $R_{\leq 10}$. Therefore, after $R_{\leq 10}$ finishes, p enters the CS after taking $O(N)$ steps. This implies that p makes $O(N)$ additional RMRs in the loop at line 112 after $R_{\leq 10}$ finishes. **(End of Case 1)**

The argument for the remaining cases is similar: it involves showing that there exists a passage by q that occurs entirely during $R_{\leq 10}$, and then applying Lemma 5.27 to show

that p makes at most $O(N)$ additional RMRs in the loop at line 112 after $R_{\leq 10}$ finishes. We summarize the cases below, but the details are omitted.

Case 2: Process p makes at least ten RMRs to $Want[q][p][1]$ for some process q. The argument for this case is nearly identical to Case 1.

Case 3: Process p makes at least ten RMRs to $Capture[q][p][2]$ for some process q. The argument for this case is nearly identical to Case 1, except that q writes $Capture[q][p][2]$ exactly once in a passage, at line 121.

Case 4: Process p makes at least ten RMRs to $Capture[q][p][1]$ for some process q. The argument for this case is nearly identical to Case 1, except that q writes $Capture[q][p][1]$ exactly once in a passage, at line 121. □

Lemma 5.32. *The algorithm in Figure 5.6 makes $\Theta(N)$ RMRs in both the DSM and CC models.*

Proof. We first prove the lemma for the CC model. A process p announces itself four times to each process in the system: twice when it first leaves the NCS in the loop at line 105, and another two times in the loop at line 108. Each of these announcements incurs $\Theta(N)$ RMRs. Choosing a ticket at line 107 also incurs $\Theta(N)$ RMRs. By Lemma 5.31, p makes $O(N)$ RMRs in the loop at line 112, and the capturing mechanism at line 119 incurs $\Theta(N)$ RMRs. Finally, in the exit protocol, p withdraws its announcement to enter the CS from every other process exactly twice. This also incurs $\Theta(N)$ RMRs. Thus the RMR complexity is $\Theta(N)$ in the CC model.

The result also follows for the DSM model by a similar argument. The only difference in the DSM model is that p makes no RMRs while in the loop at line 112, as each variable $Ticket[p]$, $Want[q][p][1]$, $Want[q][p][2]$, $Capture[q][p][1]$, and $Capture[q][p][2]$ is local to p. □

Theorem 5.33. *The algorithm in Figure 5.6 satisfies k-exclusion, starvation freedom, and FIFE when reads and writes are non-atomic. Moreover, it has RMR complexity $\Theta(N)$ in both the DSM and CC models.*

Proof. Follows from Lemmas 5.26, 5.30, 5.28, and 5.32. □

5.8 k-Exclusion with Ticket Resetting (Non-atomic RWs)

One desirable feature of the Bakery algorithm [32] that is not present in our k-exclusion algorithms is that each process resets its ticket to zero in the exit protocol. This means that if there is a period of quiescence in which all processes are in the NCS, all tickets will be reset to zero. In contrast, in our algorithms tickets necessarily grow without bound and are never reset. In this section, we modify the algorithm given in Figure 5.3 so that tickets are reset in the exit protocol. The new algorithm is presented in Figure 5.7 and works even if reads and writes are non-atomic: It satisfies k-exclusion, starvation freedom, and k-FCFS, and has RMR complexity $O(N)$. Like the algorithm in Figure 5.3, however, it does not satisfy FIFE. The doorway consists of lines 128..130, the waiting room consists of lines 131..137, and the exit protocol consists of lines 139..141.

Processes use three shared arrays to communicate with each other: *Doorway*, *Ticket*, and *Bypass*. These arrays are two dimensional in both the CC and DSM model.[3] In the DSM model the entries $Doorway[q][p]$, $Ticket[q][p]$ and $Bypass[q][p]$, for any processes p and q, are local to process p. The *Doorway* array is used by a process to announce when it is executing through the doorway of the trying protocol. In particular, when a process p starts the doorway at line 128, p announces this to each process q by setting $Doorway[p][q] = \mathsf{true}$, and after p finishes the doorway at line 130, p similarly announces this to q by setting $Doorway[p][q] = \mathsf{false}$ at line 131. The *Ticket* array is used to choose and announce tickets, which provide a rough guideline for the order in which processes are

[3]Technically, in the CC model, it suffices to use one dimensional versions of the *Doorway* and *Ticket* shared arrays. However, *Bypass* must be two dimensional in both the CC and DSM models, so it is not possible to asymptotically save any space by presenting a separate version of our algorithm for the CC model.

admitted to the CS. Specifically, the entry $Ticket[p][q]$ contains the ticket value process p has chosen and announced to process q. If p is in the NCS, then this value is 0. Lastly, the *Bypass* array is used by a process in the waiting room to determine whether other processes have *bypassed* it. Informally, we say a process q bypasses another process p in the waiting room if q executes through the exit protocol while p is in the waiting room.

To understand the algorithm at a high-level, first consider the case where $k = 1$. In this case, it turns out we can delete references to *Bypass*: line 132, the check of $Bypass[i][p]$ at line 136, and the loop at line 140. In fact, as a result of these deletions, what we arrive at is a local-spin version of Lamport's Bakery algorithm. The intuition behind Lamport's Bakery algorithm was explained in Section 5.1. To see why this version of the algorithm fails when $k = 2$ (or for any $k > 1$), consider the following scenario with processes p, q, and r. Process p enters the loop at line 134, and just before it checks entry q in $p.predecessor\_set$ (line 135), q enters the doorway. Process p then sees $Doorway[q][p]$, and thus does not remove q from $p.predecessor\_set$ at line 137. Since $k = 2$, q freely enters the CS and then returns to the NCS. Process r then enters the doorway, after which point p checks entry r in $p.predecessor\_set$. Since r is in the doorway, p cannot remove r from $p.predecessor\_set$. This scenario repeats ad infinitum. As a result, p constantly sees at least two processes in its predecessor set and is prevented from ever entering the CS. If p were able to detect q and r "bypassing" it while p was in the waiting room, then p would not have to starve. This is the motivation behind the *Bypass* array. We now provide a more complete commentary of the algorithm.

At line 129, process p chooses a ticket by determining the largest ticket currently held by some process and adding 1 to it. This roughly guarantees that p chooses a ticket that is larger than any ticket held by another active process, and places p "last in line" for entry into the CS. Since line 129 is non-atomic, other processes may execute line 129 concurrently with p, in which case it is possible that the ticket that p chooses is not in fact the largest. This does not affect the correctness of the algorithm, however. After

this, p finishes the doorway at line 130 by announcing the ticket it chose to all other processes.

At line 132, p sets $Bypass[q][p] = \mathsf{false}$ for every process q, and then at line 133, initializes a predecessor set with all other processes in the system. The predecessor set is intended to approximate the set of processes that are trying to concurrently enter the CS with p and that have priority over p to enter the CS, as in the algorithms presented in the preceding sections.

The purpose of lines 134..137 is to prevent a process p from entering the CS until enough processes are eliminated from p's predecessor set. To this end, p repeatedly tests the condition at line 136 for each process still in its predecessor set. If the condition is true for some process, then p removes that process from its predecessor set.

We now detail the purpose of each part of the condition at line 136. For any process q, if p sees $Bypass[q][p] = \mathsf{true}$, then process q bypassed p. That is, q executed through the exit protocol after p started the waiting room, which means that q does not have priority over p in p's current passage. In the second part of the condition at line 136, p checks if q is not in the doorway, and that one of the following is true: the ticket that q has announced to p is 0, in which case q is not trying to enter the CS; or that p's $(ticket, process\_id)$ pair is smaller than q's, in which case p has priority over q. If either the first or second part of the condition holds, then p removes q from its predecessor set. When the size of p's predecessor set goes below k, p enters the CS.

Note that the ordering of conditions in the second half of line 136 is important. In particular, p must check that q is not in the doorway before it checks whether the ticket that q has announced to p is 0 or that p's $(ticket, process\_id)$ pair is smaller than q's. To see why this is important, suppose that p first checks if $Ticket[q][p] = 0$, and then checks if $\neg Doorway[q][p]$ is true. If p reads $Ticket[q][p] = 0$, it is possible that q was in fact in the doorway, but had not yet announced its ticket at line 130. Then, if p sees $\neg Doorway[q][p]$, q may have already left the doorway. As a result, p may incorrectly

remove q from its predecessor set, even though q may possibly have a smaller ticket than p.

In the exit protocol (lines 139..141), p lets every process know that it no longer wants entry into the CS by announcing a ticket value of 0. After this, for each process q in the system, process p sets $Bypass[p][q] = \mathsf{true}$, but only if it is currently false. By doing this, p informs processes that may be stuck in the waiting room that p has bypassed them. The reason p sets a *Bypass* entry only when it is false is so that the algorithm works with non-atomic reads and writes. Otherwise, a situation may arise in which p executes through the CS infinitely often, but another process q stuck in the waiting room never removes p from its predecessor because q reads the *Bypass* entry concurrently with p's write.

$Bypass[p][q]$ (for each p, q) is the only shared variable whose value needs to be checked before writing (line 141). On first inspection it may seem possible that q, in a single passage, can read incorrect values from other shared variables infinitely often as a result of them being written in repeated passages. This is not the case, however, since $Bypass[p][q]$ will not be repeatedly written. Once q sees $Bypass[p][q]$ set to to true (line 136), q can remove p from $q.predecessor\_set$, and q will not read any of the other shared variables associated with p ($Doorway[p][q]$ and $Ticket[p][q]$) until q's next passage.

To see why this algorithm is local-spin in the DSM model, we observe that unbounded waiting occurs only in the loop at line 134. The variables process p accesses in this loop are all local to p, and so p makes no remote memory references while in this loop. The algorithm is also local-spin in the CC model, which we prove below in Lemma 5.46.

Also, note that unlike the preceding k-exclusion algorithm that uses only non-atomic reads and writes, none of the shared variables are duplicated. It is not necessary to duplicate *Doorway* or *Bypass*, since these variables are boolean. It is not necessary to duplicate $Ticket[q][p]$ for any process q because p first checks whether q is in the doorway (line 136) before reading $Ticket[q][p]$. To understand this in more detail, suppose that

p sees that $\neg Doorway[q][p]$ (i.e., q is not in the doorway), and then goes on to read $Ticket[q][p]$. If q concurrently writes $Ticket[q][p]$ during p's read of this variable, then the danger exists that p may read an erroneous value from $Ticket[q][p]$ and incorrectly remove q from $p.predecessor\_set$. However, since p first read that $\neg Doorway[q][p]$, it must be the case that if q concurrently writes $Ticket[q][p]$ during p's read of $Ticket[q][p]$, then q started the doorway after p finished it. Therefore, it is safe for p to remove q from $p.predecessor\_set$, and reading an incorrect value from $Ticket[q][p]$ is harmless.

The proof of correctness follows the same structure as the proofs for the earlier algorithms, however the proofs here are more complex. The proofs need to take into account that tickets can be reset, and the introduction of the bypass mechanism. These items modify the conditions under which a process waits and can potentially compromise k-exclusion.

5.8.1 Proofs

Let C_p denote the operation in which process p chooses its ticket at line 129, let A_p denote the operation in which p announces its ticket at line 130, and let EP_p denote the operation in which p executes its exit protocol at lines 139..141.

Lemma 5.34. *Let p and q be distinct processes. If $A_p \rightarrow C_q$ and $C_q \rightarrow EP_p$, then $t_p < t_q$, where t_p and t_q are the tickets chosen by p and q, respectively.*

Proof. Process p announces its ticket (line 130) before q chooses its ticket (line 129), and process q chooses its ticket (line 129) before p resets its ticket (line 139). Therefore the value that q reads from $Ticket[p][q]$ at line 129 is equal to t_p. Process q adds one to this value, and so the ticket t_q that q chooses satisfies $t_q > t_p$. □

As a corollary to the preceding lemma, we state it in the following logically equivalent form:

Corollary 5.35. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q. If $t_q \leq t_p$ and $C_q \rightarrow EP_p$, then $C_q \dashrightarrow A_p$.*

Figure 5.7: (Non-atomic RWs) k-Exclusion ticket resetting algorithm for process $p \in \{1, ..., N\}$

shared variables:
Ticket: **array**$[1..N][1..N]$ **of** $\mathbb{N}$ **init all 0**
Doorway: **array**$[1..N][1..N]$ **of boolean init all** false
Bypass: **array**$[1..N][1..N]$ **of boolean init all** false
(DSM model: $Ticket[i][p]$, $Want[i][p]$, $Bypass[i][p]$, $Doorway[i][p]$ local to p $\forall i$)

private variables:
predecessor_set: **Set of** $\mathbb{N}$
ticket: $\mathbb{N}$

```
126 loop
127 |  NCS
128 |  foreach i ∈ {1..N} do  Doorway[p][i] := true
129 |  ticket := 1 + max(Ticket[1][p], Ticket[2][p], ..., Ticket[N][p])
130 |  foreach i ∈ {1..N} do  Ticket[p][i] := ticket
131 |  foreach i ∈ {1..N} do  Doorway[p][i] := false
132 |  foreach i ∈ {1..N} do  Bypass[i][p] := false
133 |  predecessor_set := {1..N} \ {p}
134 |  while |predecessor_set| ≥ k do
135 |  |  foreach i ∈ predecessor_set do
136 |  |  |  if Bypass[i][p] ∨ (¬Doorway[i][p] ∧ (Ticket[i][p] = 0 ∨ (ticket, p) <
    |  |  |  (Ticket[i][p], i))) then
137 |  |  |  |  predecessor_set := predecessor_set \ {i}
138 |  CS
139 |  foreach i ∈ {1..N} do  Ticket[p][i] := 0
140 |  foreach i ∈ {1..N} do
141 |  |  if ¬Bypass[p][i] then  Bypass[p][i] := true
142 end loop
```

The following lemma is an analogue to Lemma 5.3 and Lemma 5.23. Its proof differs from either of the proofs for Lemma 5.3 and Lemma 5.23 as it needs to take into account the bypass mechanism, the fact that reads and writes are non-atomic, and the fact that tickets can be reset.

Lemma 5.36. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q, respectively. If $(t_q, q) < (t_p, p)$ and $C_q \rightarrow EP_p$ then p does not remove q from*

p.predecessor_set before EP_q starts.

Proof. Assume that $(t_q, q) < (t_p, p)$ and that $C_q \rightarrow EP_p$. Suppose, by way of contradiction, that p removes q from *p.predecessor_set* before EP_q starts.

Claim 5.36.1. $C_q \dashrightarrow A_p$.

Proof. The assumption that $(t_q, q) < (t_p, p)$ implies that $t_q \leq t_p$. This, the assumption that $C_q \rightarrow EP_p$, and Corollary 5.35, imply that $C_q \dashrightarrow A_p$. (Claim 5.36.1) □

Before p removes q from its predecessor set, p reads *Bypass*$[q][p]$, *Doorway*$[q][p]$, and *Ticket*$[q][p]$ (in that order) to evaluate the condition at line 136. Let R_p^B, R_p^D, and R_p^T be these reads, respectively.

Claim 5.36.2. *R_p^B returns false.*

Proof. The only processes that write into *Bypass*$[q][p]$ are p and q. Process p sets *Bypass*$[q][p]$ = false at line 132 (denote this write operation W_p^B) and process q may set *Bypass*$[q][p]$ = true at line 141 of the exit protocol. There are no other places where *Bypass*$[q][p]$ is written. By inspection of the algorithm, $W_p^B \rightarrow R_p^B$.

Let W_q^B be any write operation by q into *Bypass*$[q][p]$. It remains to show that either $W_q^B \rightarrow W_p^B$ or $R_p^B \rightarrow W_q^B$. (To see why this suffices, note that if this is the case for all writes W_q^B by q into *Bypass*$[q][p]$, there is a well-defined last write into *Bypass*$[q][p]$ that precedes R_p^B, namely W_p^B, which sets this variable false, and there are no writes into *Bypass*$[q][p]$ concurrent with R_p^B.)

As we observed, q only writes *Bypass*$[q][p]$ in its exit protocol. So, to prove that either $W_q^B \rightarrow W_p^B$ or $R_p^B \rightarrow W_q^B$ it suffices to prove that: (1) if EP'_q is q's execution of the exit protocol in its preceding passage, then $EP'_q \rightarrow W_p^B$, and (2) $R_p^B \rightarrow EP_q$.

For (1) we have that, by the structure of the algorithm and Claim 5.36.1, $EP'_q \rightarrow C_q \dashrightarrow A_p \rightarrow W_p^B$, from which it follows that $EP'_q \rightarrow W_p^B$.

For (2) we have that, by the structure of the algorithm, R_p^B ends before p removes q from *p.predecessor_set*, which, by the hypothesis of this lemma, happens before EP_q starts. Thus, $R_q^B \rightarrow EP_q$. (Claim 5.36.2) □

By Claim 5.36.2, and lines 136..137, for p to remove q from *p.predecessor_set*, R_p^D must return false, and R_p^T must return 0 or some value t such that $(t_p, p) < (t, q)$.

Let W_q^D and $W_q^{D^+}$ be the successive write operations in which q writes, respectively, true and false into *Doorway*$[q][p]$ at lines 128 and 131.

Claim 5.36.3. $W_q^{D^+} \dashrightarrow R_p^D$

Proof. By Claim 5.36.1 and the structure of the algorithm, $W_q^D \rightarrow C_q \dashrightarrow A_p \rightarrow R_p^D$, and so $W_q^D \rightarrow R_p^D$. If $R_p^D \rightarrow W_q^{D^+}$, then $W_q^D \rightarrow R_p^D \rightarrow W_q^{D^+}$. That is, R_p^D occurs entirely between two successive writes into *Doorway*$[q][p]$, and hence must return the value written by W_q^D, which is true. This, however, contradicts that R_p^D returns false. Therefore it is not the case that $R_p^D \rightarrow W_q^{D^+}$, and so, $W_q^{D^+} \dashrightarrow R_p^D$. (Claim 5.36.3) □

Let W_q^T and $W_q^{T^+}$ be the successive write operations in which q writes t_q and 0, respectively, into *Ticket*$[q][p]$ at lines 130 and 139. By inspection of the order of operations on line 136, we have that $R_p^D \rightarrow R_p^T$. This and Claim 5.36.3 imply that $W_q^T \rightarrow W_q^{D^+} \dashrightarrow R_p^D \rightarrow R_p^T$, and so $W_q^T \rightarrow R_p^T$.

R_p^T finishes before process p removes q from *p.predecessor_set*, and by assumption, process p removes q from *p.predecessor_set* before EP_q starts. Therefore $R_p^T \rightarrow EP_q$. Furthermore, $W_q^{T^+} \subseteq EP_q$, and so $R_p^T \rightarrow W_q^{T^+}$. This and the fact that $W_q^T \rightarrow R_p^T$ (established in the preceding paragraph) imply that $W_q^T \rightarrow R_p^T \rightarrow W_q^{T^+}$. That is, R_p^T occurs entirely between two successive writes to *Ticket*$[q][p]$, and hence must return the value written by W_q^T, which is t_q. Processes always choose non-zero tickets, so $t_q \neq 0$.

Recall that R_p^B, R_p^D, and R_p^T are p's reads of *Bypass*$[q][p]$, *Doorway*$[q][p]$, and *Ticket*$[q][p]$ when p evaluates the condition at line 136 before p removes q from *p.predecessor_set*. By Claim 5.36.2, R_p^B returns false. So, by the condition at line 136,

R_p^D must return false, and R_p^T must return zero or a value t such that $(t_p, p) < (t, q)$. We established in the preceding paragraph that R_p^T returns $t_q \neq 0$. Therefore $(t_p, p) < (t_q, q)$, contrary to the assumption that $(t_q, q) < (t_p, p)$. □

We now prove that k-exclusion holds, which relies on Lemma 5.36. The proof follows the same structure as the proofs of k-exclusion for the simpler algorithms. The added complexity due to ticket resetting and the non-atomicity of reads and writes is taken care of in the proof of Lemma 5.36.

Lemma 5.37. *The algorithm in Figure 5.7 satisfies k-exclusion.*

Proof. Suppose, by way of contradiction, that $k + 1$ processes are in the CS concurrently. Let Y be this set of processes, and let p be the process in Y with the largest $(ticket, process\_id)$ pair. Moreover, as all the processes in Y are in the CS concurrently, every process in $Y \setminus \{p\}$ finishes line 129 before p starts the exit protocol. That is, for any $q \in Y \setminus \{p\}$, $C_q \rightarrow EP_p$. Thus, by Lemma 5.36, when p executes the trying protocol, p does not remove any process in Y from $p.predecessor\_set$ until some process in Y starts executing the exit protocol. This implies that the size of p's predecessor set is at least k until some process in Y leaves the CS. Thus p cannot enter the CS until some process in Y leaves the CS, which contradicts that p is in the CS concurrently with all processes in Y. □

The following technical lemma will be used to prove that the algorithm satisfies starvation freedom, and has $O(N)$ RMR complexity in the CC model.

Lemma 5.38. *Let p and q be distinct processes, and assume that q starts executing a passage P_q after p finishes writing* false *to $Bypass[q][p]$ at line 132. There exists at most one read R_p by p to $Bypass[q][p]$ at line 136 in p's current passage such that $P_q \rightarrow R_p$.*

Proof. If there exists no read R_p by p to $Bypass[q][p]$ such that $P_q \rightarrow R_p$, then we're done. So assume that such a read exists, and that R_p is the first such read by p.

Let W_p^0 be the operation in which p writes false to $Bypass[q][p]$ at line 132, and let R_q be q's read of $Bypass[q][p]$ at line 141. R_q either returns true or false.

Case 1: R_q returns false. By inspection of line 141, q writes true to $Bypass[q][p]$ immediately after R_q. Let W_q be this write operation, which is part of passage P_q. By assumption, $P_q \to R_p$, and so $W_q \to R_p$. We will now prove that R_p returns true. Once we do so, the following holds. After R_p returns true, p removes q from $p.predecessor\_set$, and there cannot be any more reads by p to $Bypass[q][p]$ in p's current passage, proving the lemma. It remains to prove that R_p returns true.

$W_q \to R_p$, and W_q writes true to $Bypass[q][p]$. Thus, to prove that R_p returns true, we will show that for any write $\hat{W}$ to $Bypass[q][p]$ other than W_q, either $\hat{W} \to W_q$ or $R_p \to \hat{W}$. Processes p and q are the only processes to write $Bypass[q][p]$. We consider these cases separately:

Case 1a: $\hat{W}$ is a write by process p. By assumption, $W_p^0 \to P_q$. Since W_q is part of P_q, $W_p^0 \to W_q$. If p does not write $Bypass[q][p]$ after W_p^0, then for every write $\hat{W}$ by p to $Bypass[q][p]$, $\hat{W} \to W_q$, and we're done. So assume that p does write $Bypass[q][p]$ after W_p^0. Let W_p^1 be the first such write after W_p^0. Process p only writes $Bypass[q][p]$ once per passage at line 132, and so W_p^1 occurs in a later passage than W_p^0. W_p^0 and R_p occur in the same passage, and so W_p^1 occurs in a later passage than R_p, i.e., $R_p \to W_p^1$. Thus, for every write $\hat{W}$ by p to $Bypass[q][p]$, either $\hat{W} \to W_q$ (if $\hat{W}$ is W_p^0 or an earlier write) or $R_p \to \hat{W}$ (if $\hat{W}$ is W_p^1 or a later write).

Case 1b: $\hat{W}$ is a write by process q. If q does not write $Bypass[q][p]$ after W_q, then for every write $\hat{W}$ by q to $Bypass[q][p]$ other than W_q, $\hat{W} \to W_q$, and we're done. So assume that there is a write W_q^1 by q to $Bypass[q][p]$ that occurs after W_q, and assume that W_q^1 is the first such write by q.

Claim 5.38.1. $R_p \to W_q^1$.

Proof. Suppose, for contradiction, that $W_q^1 \dashrightarrow R_p$. Process q writes $Bypass[q][p]$ at most

once per passage at line 141, and so W_q^1 occurs in a later passage than W_q. Let R_q^1 be the read by q of $Bypass[q][p]$ that immediately precedes W_q^1 and occurs in the same passage as W_q^1. By inspection of line 141, R_q^1 must return false for the write W_q^1 to happen. We will now prove that R_q^1 returns true, which contradicts that it must return false.

By assumption $W_p^0 \to P_q$, and W_q is part of P_q, so $W_p^0 \to W_q$. Hence, by the structure of the algorithm, $W_p^0 \to W_q \to R_q^1$. We will show that for every write $\bar{W}$ to $Bypass[q][p]$ other than W_q, either $\bar{W} \to W_q$ or $R_q^1 \to \bar{W}$. This proves that R_q^1 returns the same value as written by W_q, i.e., true. So it remains to prove that for every write $\bar{W}$ other than W_q, either $\bar{W} \to W_q$ or $R_q^1 \to \bar{W}$.

Process p and q are the only processes to write $Bypass[q][p]$. First consider writes by process q. W_q^1 is the first write to $Bypass[q][p]$ after W_q, and it occurs immediately after R_q^1. Thus, for every write $\bar{W}$ to $Bypass[q][p]$ by q other than W_q, either $\bar{W} \to W_q$ (if $\bar{W}$ occurs before W_q) or $R_q^1 \to \bar{W}$ (if $\bar{W}$ is W_q^1 or a later write), as desired.

Next consider writes $\bar{W}$ by process p. If p does not write $Bypass[q][p]$ after W_p^0, then by $W_p^0 \to W_q$, it follows that for every write $\bar{W}$ by p to $Bypass[q][p]$, $\bar{W} \to W_q$, and we're done. So assume that p does write $Bypass[q][p]$ after W_p^0. Let W_p^1 be the first such write. Process p only writes $Bypass[q][p]$ once per passage at line 132, and so W_p^1 must occur in a later passage than W_p^0. Also, W_p^0 and R_p occur in the same passage, and so W_p^1 must occur in a later passage than R_p, i.e., $R_p \to W_p^1$. By our assumption that $W_q^1 \dashrightarrow R_p$, and the structure of the algorithm, we thus have that $R_q^1 \to W_q^1 \dashrightarrow R_p \to W_p^1$, and so $R_q^1 \to W_p^1$. This and $W_p^0 \to W_q$ imply that for every write $\bar{W}$ to $Bypass[q][p]$ by p, either $\bar{W} \to W_q$ (if $\bar{W}$ is W_p^0 or an earlier write) or $R_q^1 \to \bar{W}$ (if $\bar{W}$ is W_p^1 or a later write), as desired. □

Recall that W_q^1 is the first write to $Bypass[q][p]$ by q after W_q. Thus, by Claim 5.38.1, for every write $\hat{W}$ by q to $Bypass[q][p]$ other than W_q, either $\hat{W} \to W_q$ (if $\hat{W}$ occurs before W_q) or $R_p \to \hat{W}$ (if $\hat{W}$ is W_q^1 or a later write).

Case 2: R_q returns true. By assumption, $P_q \to R_p$, and R_q is part of P_q, so

$R_q \to R_p$. We will now prove that R_p returns true. After R_p returns true, p removes q from $p.predecessor\_set$, and there cannot be any more reads by p to $Bypass[q][p]$ in p's current passage, proving the lemma. It remains to prove that R_p returns true.

$R_q \to R_p$, and by assumption of this case, R_q returns true. To prove that R_p returns true, we will show that for any write $\hat{W}$ to $Bypass[q][p]$, either $\hat{W} \to R_q$ or $R_p \to \hat{W}$. Processes p and q are the only processes to write $Bypass[q][p]$. We consider these cases separately:

Case 2a: $\hat{W}$ is a write by process p. By assumption, $W_p^0 \to P_q$. Since R_q is part of P_q, $W_p^0 \to R_q$. If p does not write $Bypass[q][p]$ after W_p^0, then for every write $\hat{W}$ by p to $Bypass[q][p]$, $\hat{W} \to R_q$, and we're done. So assume that p does write $Bypass[q][p]$ after W_p^0. Let W_p^1 be the first such write after W_p^0. Process p only writes $Bypass[q][p]$ once per passage at line 132, and so W_p^1 occurs in a later passage than W_p^0. W_p^0 and R_p occur in the same passage, and so W_p^1 occurs in a later passage than R_p, i.e., $R_p \to W_p^1$. Thus, for every write $\hat{W}$ by p to $Bypass[q][p]$, either $\hat{W} \to R_q$ (if $\hat{W}$ is W_p^0 or an earlier write) or $R_p \to \hat{W}$ (if $\hat{W}$ is W_p^1 or a later write).

Case 2b: $\hat{W}$ is a write by process q. If q does not write $Bypass[q][p]$ after R_q, then for every write $\hat{W}$ by q to $Bypass[q][p]$, $\hat{W} \to R_q$, and we're done. So assume that there is a write W_q^1 by q to $Bypass[q][p]$ that occurs after R_q, and assume that W_q^1 is the first such write by q.

Claim 5.38.2. $R_p \to W_q^1$.

Proof. Suppose, for contradiction, that $W_q^1 \dashrightarrow R_p$. Process q writes $Bypass[q][p]$ at most once per passage at line 141, and so W_q^1 occurs in a later passage than R_q. Let R_q^1 be the read by q of $Bypass[q][p]$ that immediately precedes W_q^1 and occurs in the same passage as W_q^1. By inspection of line 141, R_q^1 must return false for the write W_q^1 to happen. We will now prove that R_q^1 returns true, which contradicts that it must return false.

By assumption $W_p^0 \to P_q$, and R_q is part of P_q, so $W_p^0 \to R_q$. Hence, by the structure

of the algorithm, $W_p^0 \rightarrow R_q \rightarrow R_q^1$. We will show that for every write $\bar{W}$ to $Bypass[q][p]$, either $\bar{W} \rightarrow R_q$ or $R_q^1 \rightarrow \bar{W}$. This proves that R_q^1 returns the same value as R_q, i.e., true. So it remains to prove that for every write $\bar{W}$, either $\bar{W} \rightarrow R_q$ or $R_q^1 \rightarrow \bar{W}$.

Process p and q are the only processes to write $Bypass[q][p]$. First consider writes by process q. W_q^1 is the first write to $Bypass[q][p]$ after R_q, and it occurs immediately after R_q^1. Thus, for every write $\bar{W}$ to $Bypass[q][p]$ by q, either $\bar{W} \rightarrow R_q$ (if $\bar{W}$ occurs before R_q) or $R_q^1 \rightarrow \bar{W}$ (if $\bar{W}$ is W_q^1 or a later write), as desired.

Next consider writes $\bar{W}$ by process p. If p does not write $Bypass[q][p]$ after W_p^0, then by $W_p^0 \rightarrow R_q$, it follows that for every write $\bar{W}$ by p to $Bypass[q][p]$, $\bar{W} \rightarrow R_q$, and we're done. So assume that p does write $Bypass[q][p]$ after W_p^0. Let W_p^1 be the first such write. Process p only writes $Bypass[q][p]$ once per passage at line 132, and so W_p^1 must occur in a later passage than W_p^0. Also, W_p^0 and R_p occur in the same passage, and so W_p^1 must occur in a later passage than R_p, i.e., $R_p \rightarrow W_p^1$. By our assumption that $W_q^1 \dashrightarrow R_p$, and the structure of the algorithm, we thus have that $R_q^1 \rightarrow W_q^1 \dashrightarrow R_p \rightarrow W_p^1$, and so $R_q^1 \rightarrow W_p^1$. This and $W_p^0 \rightarrow R_q$ imply that for every write $\bar{W}$ to $Bypass[q][p]$ by p, either $\bar{W} \rightarrow R_q$ (if $\bar{W}$ is W_p^0 or an earlier write) or $R_q^1 \rightarrow \bar{W}$ (if $\bar{W}$ is W_p^1 or a later write), as desired. □

Recall that W_q^1 is the first write to $Bypass[q][p]$ by q after R_q. Thus, by Claim 5.38.2, for every write $\hat{W}$ by q to $Bypass[q][p]$, either $\hat{W} \rightarrow R_q$ (if $\hat{W}$ occurs before R_q) or $R_p \rightarrow \hat{W}$ (if $\hat{W}$ is W_q^1 or a later write). □

The following two corollaries to Lemma 5.38 follow as a result of the ordering of read operations at line 136. They will also used to prove that the algorithm has $O(N)$ RMR complexity in the CC model.

Corollary 5.39. *Let p and q be distinct processes, and assume that q starts executing a passage P_q after p finishes writing* false *to $Bypass[q][p]$ at line 132. There exists at most two reads R_p by p to $Doorway[q][p]$ at line 136 in p's current passage such that $P_q \rightarrow R_p$.*

Corollary 5.40. *Let p and q be distinct processes, and assume that q starts executing a passage P_q after p finishes writing* false *to $Bypass[q][p]$ at line 132. There exists at most four reads R_p by p to $Ticket[q][p]$ at line 136 in p's current passage such that $P_q \rightarrow R_p$.*

Next, we prove starvation freedom. The proof below is somewhat more intricate than the proof of starvation freedom in Lemma 5.5 as a result of tickets being reset.

Lemma 5.41. *The algorithm in Figure 5.7 satisfies starvation freedom.*

Proof. Suppose, by way of contradiction, that starvation freedom does not hold. Let Y be the set of live processes that are stuck forever in the trying protocol, and let p be the process in Y with the smallest $(ticket, process\_id)$ pair.

Below we show that p eventually removes all live processes from its predecessor set, which implies that eventually $p.predecessor\_set$ contains only faulty processes. As there are at most $k-1$ faulty processes, this means that p eventually executes past the loop at line 134 and enters the CS, contradicting that it is stuck forever in the trying protocol.

Process p is the process in Y with the smallest $(ticket, process\_id)$ pair, and so by inspection of the code at line 136 it follows that p eventually removes all processes in $Y \setminus \{p\}$ from its predecessor set.

It remains to show that p also removes from its predecessor set all the live processes that are not eventually stuck forever in the trying protocol. We can split these processes into two groups: live processes that execute through the CS infinitely often, and live processes that eventually remain in the NCS forever. We consider the latter group first, and let r_{NCS} be any such process. Initially, $Doorway[r_{NCS}][p] =$ false and $Ticket[r_{NCS}][p] = 0$. After r_{NCS}'s last passage, $Doorway[r_{NCS}][p] =$ false and $Ticket[r_{NCS}][p] = 0$ are also true. By inspection of the code, it follows that p eventually evaluates the condition at line 136 for iteration $i = r_{NCS}$ to be true and removes r_{NCS} from $p.predecessor\_set$.

We next consider the group of live processes that execute through the CS infinitely often. Let r_{IO} be any such process. Let W_p^B be p's write of false to $Bypass[r_{IO}][p]$ at

line 132, and let $P_{r_{IO}}$ be a passage of r_{IO} such that $W_p^B \rightarrow P_{r_{IO}}$. (This passage by r_{IO} exists since r_{IO} executes through the CS infinitely often.) By Lemma 5.38 (with r_{IO} taking on the role of q), there exists at most one read R_p by p to $Bypass[r_{IO}][p]$ such that $P_{r_{IO}} \rightarrow R_p$. By assumption, however, p does not crash and is stuck forever in the loop at line 134, and so process p reads $Bypass[i][p]$ at line 136 infinitely often for any process i in its predecessor set. The preceding two sentences imply that p eventually removes r_{IO} from its predecessor set.

Thus p eventually removes from $p.predecessor\_set$ every live process r_{IO} that executes through the CS infinitely often. We also established that p eventually removes from $p.predecessor\_set$ every live process r_{NCS} that is eventually in the NCS forever, and all processes in $Y \setminus \{p\}$. Hence the only processes that p does not remove from $p.predecessor\_set$ are faulty processes. There are at most $k-1$ such processes, and so p eventually executes past the loop at line 134 and enters the CS, contrary to the assumption that p is stuck in the trying protocol forever. □

The algorithm satisfies k-FCFS, but not the stronger FIFE property. To see why FIFE is not satisfied, consider the following execution for $k = 2$ with processes p_1, p_2, p_3, and p_4. Process p_1 leaves the NCS and finishes executing the doorway, after which it temporarily stops taking steps. Process p_2 then leaves the NCS, executes the doorway and the waiting room, and enters the CS. After this, p_3 and p_4 start executing the doorway. For FIFE to be satisfied, p_1 should now be able to enter the CS in a bounded number of its own steps. However, this is not the case, because p_1 cannot remove either p_3 or p_4 from $p_1.predecessor\_set$ while p_3 and p_4 are in the doorway, thus preventing p_1 from advancing past the loop at line 134.

We now prove that k-FCFS is satisfied.

Lemma 5.42. *The algorithm in Figure 5.7 satisfies the k-FCFS property.*

Proof. Let Y be a set of processes such that $|Y| = k$, and assume all processes in Y

finish the doorway before a process p starts the doorway (our first assumption). Further, suppose, by way of contradiction, that p enters the CS before every process in Y (our second assumption). We have that $\forall q \in Y$ (i) $A_q \rightarrow C_p$ (by our first assumption); (ii) $C_q \rightarrow EP_p$ (by our first assumption); and (iii) $C_p \rightarrow EP_q$ (by our second assumption). By (i), (iii), and Lemma 5.34 we know that every process in Y chooses a ticket strictly smaller than the ticket that p chooses. By this, (ii), and Lemma 5.36, it follows that during p's execution of the trying protocol, p does not remove any process in Y from *p.predecessor_set* until at least one process in Y completes the CS. This implies that the size of *p.predecessor_set* will be at least k until some process in Y executes through the CS. Process p does not enter the CS until the size of its predecessor set is less than k, and so p does not enter the CS until some process in Y enters the CS. This contradicts that p enters the CS before every process in Y. □

Lemma 5.43. *Let p and q be distinct processes. In the CC model, while p is executing the loop at line 134, process p makes at most eight RMRs reading $Doorway[q][p]$ at line 136.*

Proof. Suppose, by way of contradiction, that p makes at least nine RMRs while reading $Doorway[q][p]$ at line 136. Recall that in the CC model a process makes an RMR when it reads a variable for the first time, writes a variable, and whenever a process reads a variable for the first time after another process has written that variable. Therefore, the reason for p's first RMR to $Doorway[q][p]$ at line 136 could be because p does not have $Doorway[q][p]$ in its cache, but the eight subsequent RMRs must be as a result of some process invalidating p's cached copy of $Doorway[q][p]$ eight times. Processes q is the only process that writes $Doorway[q][p]$ (at lines 128 and 131). Hence, there must exist eight writes by q to $Doorway[q][p]$, each of which invalidates a copy of $Doorway[q][p]$ in p's cache. Process q writes $Doorway[q][p]$ at most twice per passage, and so q must start executing a passage after p finishes its first RMR to $Doorway[q][p]$, and q must finish that passage before p starts its seventh RMR to $Doorway[q][p]$. Thus, there exists a passage

P_q by q that starts after p finishes writing false to $Bypass[q][p]$ at line 132, and P_q finishes before p starts its seventh RMR. By Corollary 5.39, there exists at most two reads R_p by p to $Doorway[q][p]$ such that $P_q \rightarrow R_p$. This means that p's seventh and eighth RMRs to $Doorway[q][p]$ are the last two RMRs to $Doorway[q][p]$ that p can make in its current passage, which contradicts that p makes at least nine RMRs while reading $Doorway[q][p]$ at line 136. □

We omit the proofs of the following two lemmas, as they are very similar to the proof of Lemma 5.43.

Lemma 5.44. *Let p and q be distinct processes. In the CC model, while p is executing the loop at line 134, process p makes at most ten RMRs reading $Ticket[q][p]$ at line 136.*

Lemma 5.45. *Let p and q be distinct processes. In the CC model, while p is executing the loop at line 134, process p makes at most five RMRs reading $Bypass[q][p]$ at line 136.*

Lemma 5.46. *A process p makes $\Theta(N)$ RMRs in a passage of the algorithm in Figure 5.7 in both the DSM and CC models.*

Proof. We first do the proof for the CC model. The only shared variables that a process p accesses in the loop at line 134 are $Bypass[i][p]$, $Ticket[i][p]$ and $Doorway[i][p]$ (for each $i = 1..N$). This, Lemmas 5.43, 5.44, 5.45, and the fact that there are N processes, imply that p makes at most $O(N)$ RMRs while p is executing the loop at line 134. Elsewhere in the algorithm, it is easily seen by inspection that p makes $\Theta(N)$ RMRs. Thus the RMR complexity is $\Theta(N)$ in the CC model.

The result also follows for the DSM model by observing that p makes no RMRs while in the loop at line 134, as the variables $Bypass[i][p]$, $Ticket[i][p]$, and $Doorway[i][p]$ (for each $i = 1..N$) are local to p. □

Theorem 5.47. *The algorithm in Figure 5.7 satisfies k-exclusion, starvation freedom, and k-FCFS. Moreover, it has RMR complexity $\Theta(N)$ in the DSM and CC models.*

Proof. The result follows from Lemmas 5.37, 5.41, 5.42, and 5.46. □

5.9 k-Exclusion With Ticket Resetting and FIFE (Atomic RWs)

Our goal in this section is to modify the algorithm in Figure 5.7 so that it also satisfies FIFE. The modifications required turn out to be non-trivial. To simplify things, we first present a version of the algorithm, in Figure 5.8, that uses atomic reads and writes. An embellished version that works even with non-atomic reads and writes is presented in Section 5.10.

The algorithm in Figure 5.8 satisfies k-exclusion, starvation freedom, FIFE, and resets tickets in the exit protocol. The doorway consists of lines 145..153, the waiting room consists of lines 154..164, and the exit protocol consists of lines 166..167.

Note that on line 167 we can now "blind-set" $Bypass[p][i]$ to true, while at the corresponding point in he algorithm of Figure 5.7 (see line 141) we set $Bypass[p][i]$ to true only after verifying that its present value is false. This simplification is due to the fact that we are now assuming that reads and writes are atomic. As we explained in the previous section, blind-setting $Bypass[p][i]$ to true on line 141 can lead to a violation of starvation freedom when reads and writes are non-atomic. However, when atomic reads/writes are used, this problem cannot occur. We now explain how FIFE is implemented.

The FIFE property states that if a process p doorway-precedes a process q, and q enters the CS, then p can enter the CS in a bounded number of its own steps. Intuitively, the algorithm in Figure 5.8 satisfies FIFE by ensuring that a process that is about to enter the CS first "captures" processes that doorway-preceded it. If a process detects that it has been captured, it is free to enter the CS. At a high-level, this idea is similar to that used by the algorithm we described in Section 5.6, which satisfies FIFE but does not reset tickets in the exit protocol. The capturing mechanism is considerably more

complicated now that tickets are to be reset in each passage, as we explain below.

To implement the capturing mechanism, we introduce a new shared two-dimensional array, $Capture[i][j]$. Whenever a process q attempts to capture a process p, q writes the ticket that q chose in the doorway to $Capture[q][p]$ (line 164). In the waiting room, process p monitors whether it has been captured by q by checking if the entry $Capture[q][p]$ stores a value that is larger than the ticket that p chose in the doorway (line 163). If so, then p is free to enter the CS. This simple mechanism ensures FIFE is satisfied, since if p doorway-precedes q, then q chooses and writes to $Capture[q][p]$ a ticket that is strictly larger than the ticket p chooses.

Since tickets are reset in the exit protocol, each process p needs to also reset the entries $Capture[j][p]$ (for all j) somewhere in the algorithm. If not, then it is possible for p to think that it has been captured in some passage, when in fact it was captured in an earlier passage. It may appear that the natural place to reset these entries is the exit protocol, where the tickets are reset; this, however, can lead to problems: Process q may write to $Capture[q][p]$ after p has finished the exit protocol, thus leading to a situation in which p starts a new passage with the $Capture[q][p]$ entry not reset. So, in our algorithm, process p actually resets $Capture[q][p]$ in the doorway at line 149. In this manner, p resets $Capture[q][p]$ in the same passage in which it checks whether it has been captured by q.

The capturing mechanism as we have described it so far is similar to that used in the algorithm in Figure 5.5, except for the resetting of variables $Capture[q][p]$ (see line 149). This resetting, as we have seen, is needed so that p does not mistakenly consider itself captured by q if p happens to choose a small ticket in a subsequent passage — an eventuality that is possible now that tickets are reset to 0 after each passage.

The resetting of tickets necessitates another, perhaps subtler, change to the capturing mechanism. In the algorithm in Figure 5.5 it was perfectly fine for a process to attempt to capture all other processes (see the loop on line 99). It was harmless for a process to attempt to capture even processes whose doorway executions did not precede its own.

Now that tickets are reset in each passage, this is no longer the case. A process must now refrain from attempting to capture certain processes: note that the loop on line 164 is not executed for processes in the set *ignore_set*, which was previously computed on lines 151..153. If a process does not refrain from attempting to capture the processes in *ignore_set*, a violation of k-exclusion may result.

To gain some intuition about the problem and its solution, suppose, temporarily, that a process attempts to capture all processes; i.e., assume that the loop on line 164 is executed for all $i \in \{1..N\}$. In this version of the algorithm, the possibility exists that a process attempts to capture another process even when this is not required to satisfy FIFE and, in doing so, can cause a violation of k-exclusion. First we consider a simple scenario for $k > 1$ that appears benign and does not lead to a violation of k-exclusion. We then show how to modify the scenario so that k-exclusion is violated. Suppose processes p and q execute concurrently through the doorway and end up choosing tickets so that p gets a smaller ticket than q. Process p resets $Capture[q][p]$ (line 149), after which process q advances into the waiting room, captures p (line 164), and finally q enters the CS. (Process q can enter the CS even though it has a larger ticket than p since $k > 1$.) When process p enters the loop at line 158, it can also enter the CS: p will either detect that it is captured, or see that it has a smaller ticket than q. In this scenario, the fact that q captured p, despite being unnecessary for FIFE, is harmless and does not lead to a violation of k-exclusion, since $k > 1$. However, by introducing processes r_1 and r_2 into this scenario and having them execute their passages concurrently with p and q, the following violation of k-exclusion (for $k = 2$) can happen:

1. Process p and q start choosing a ticket concurrently.

2. Process q chooses a ticket (line 147) and finishes the doorway.

3. Process p chooses a ticket smaller than q's ticket (line 147), but does not yet announce it (line 148). (Process p chooses a smaller ticket than q as a result of

other processes that were still in the CS while q was choosing its ticket, but then returned to the NCS before p read the ticket values associated with those processes.)

4. Process q reaches the loop at line 164, but does not yet attempt to capture any processes. (Process q can advance past the loop at line 158 while p is still in the doorway since $k = 2$.)

5. Processes r_1 and r_2 enter the doorway and start choosing their tickets (line 147). They read p's ticket, which is currently 0, but they do not yet read q's ticket.

6. Process p announces its ticket, resets $Capture[q][p]$, and finishes the doorway.

7. Process q writes its ticket, which is larger than p's ticket, to $Capture[q][p]$ at line 164, thus capturing p. Process q then executes through the CS and the exit protocol, where q resets its ticket (line 166), after which q returns to the NCS.

8. Processes r_1 and r_2 finish the doorway, each choosing a smaller ticket than the tickets chosen by p or q. (The reason for this is that when r_1 and r_2 read the tickets announced by p and q, they read the value 0.)

9. Process p detects that it has been captured (line 163) and enters the CS.

10. Processes r_1 and r_2 eliminate from their predecessor sets all processes in the NCS (i.e., processes with ticket value 0), including q, and all processes with higher numbered tickets (line 160), including p. This leaves at most $1 < k = 2$ element in the predecessor sets of r_1 and r_2. In particular, r_1 possibly has r_2 in its predecessor set, and r_2 possibly has r_1 in its predecessor set.

11. Processes r_1 and r_2 enter the CS.

12. There are now 3 processes in the CS, and k-exclusion (for $k = 2$) is violated.

By introducing processes $r_1, r_2, \ldots, r_k$, this scenario can be generalized to demonstrate a violation of k-exclusion for any $k > 1$

The reason that the above scenario leads to a violation of k-exclusion is because q resets its ticket entries in the exit protocol. If q did not do this, then r_1 and r_2 would have chosen tickets larger than both p and q, and k-exclusion would be satisfied.

To fix this problem, either we need to prevent q from resetting its ticket entries before r_1 and r_2 have had a chance to read q's ticket, or we need to ensure that r_1 and r_2 do not read p's ticket before p has announced it. Unfortunately, these solutions are not viable. The former solution is not viable, because it would require q to wait for an indeterminate amount of time before executing its exit protocol, and the latter solution is not viable, because r_1 and r_2 have no control over when p announces its ticket. We can, however, do something that achieves a similar result: we can ensure that q attempts to capture p only under circumstances necessary to satisfy FIFE.

Ideally, we would like for process q to attempt to capture p if and only if p finishes the doorway before q starts the doorway. This is not possible, but the following approximation to this ideal is sufficient for our purposes. We modify the algorithm so that: (i) if p finishes the doorway before q starts the doorway, then q attempts to capture p; and (ii) if q attempts to capture p, then p finishes announcing its ticket (line 148) before q starts choosing its ticket (line 147). This feature of the algorithm does not prevent all unnecessary capture attempts, but it does prevent the ones that lead to the bad scenario that we outlined, and it turns out that this is enough to restore k-exclusion. Revisiting the scenario, we see that for q to attempt to capture p, p must announce its ticket before q chooses its ticket. Since processes r_1 and r_2 leave the NCS after q finishes the doorway, p will have already announced its ticket when processes r_1 and r_2 try reading it. Therefore, r_1 and r_2 will choose a larger ticket than p chooses, and k-exclusion will not be violated.

Figure 5.8: (Atomic RWs) k-Exclusion with FIFE and ticket resetting for process $p \in \{1, ..., N\}$

shared variables:
Ticket: **array**$[1..N][1..N]$ **of** $\mathbb{N}$ **init all 0**
Doorway: **array**$[1..N][1..N]$ **of boolean init all** false
Bypass: **array**$[1..N][1..N]$ **of boolean init all** false
Capture: **array**$[1..N][1..N]$ **of** $\mathbb{N}$ **init all 0**
DWYBypass: **array**$[1..N][1..N]$ **of boolean init all** false

(DSM model: *Ticket*$[i][p]$, *Capture*$[i][p]$, *Bypass*$[i][p]$, *Doorway*$[i][p]$ local to p $\forall i$)

private variables:
predecessor_set, *ignore_set*: **Set of** $\mathbb{N}$
ticket: $\mathbb{N}$
captured: **boolean**

```
143 loop
144     NCS
145     foreach i ∈ {1..N} do  Doorway[p][i] := true
146     foreach i ∈ {1..N} do  DWYBypass[i][p] := false
147     ticket := 1 + max(Ticket[1][p], Ticket[2][p], ..., Ticket[N][p])
148     foreach i ∈ {1..N} do  Ticket[p][i] := ticket
149     foreach i ∈ {1..N} do  Capture[i][p] := 0
150     foreach i ∈ {1..N} do  DWYBypass[p][i] := true
151     ignore_set := ∅
152     foreach i ∈ {1..N} do
153         if Doorway[i][p] ∨ DWYBypass[i][p] then  ignore_set := ignore_set ∪ {i}
154     foreach i ∈ {1..N} do  Doorway[p][i] := false
155     foreach i ∈ {1..N} do  Bypass[i][p] := false
156     predecessor_set := {1..N} \ {p}
157     captured := false
158     while |predecessor_set| ≥ k ∧ ¬captured do
159         foreach i ∈ predecessor_set do
160             if Bypass[i][p] ∨ (¬Doorway[i][p] ∧ (Ticket[i][p] = 0 ∨ (ticket, p) <
                (Ticket[i][p], i))) then
161                 predecessor_set := predecessor_set \ {i}
162         foreach i ∈ {1..N} do
163             if ticket < Capture[i][p] then  captured := true
164     foreach i ∈ {1..N} \ ignore_set do  Capture[p][i] := ticket
165     CS
166     foreach i ∈ {1..N} do  Ticket[p][i] := 0
167     foreach i ∈ {1..N} do  Bypass[p][i] := true
168 end loop
```

We implement the preceding feature by using the shared array *DWYBypass* and the private variable *ignore_set*. When q starts the doorway, process q sets $DWYBypass[p][q]$ to be false at line 146. The only other time that $DWYBypass[p][q]$ is written is when p sets it to be true at line 150, near the end of the doorway. Before q finishes the doorway, it checks the value of $Doorway[p][q]$ and $DWYBypass[p][q]$ at line 153 to test whether p concurrently executed the "main part" of the doorway with q. If q reads true in $Doorway[p][q]$, then p is still in the doorway. If q reads true in $DWYBypass[p][q]$, then p must have been in the doorway at least once while q was executing it. In either of these cases, q adds p to $q.ignore\_set$, and then refrains from attempting to capture the processes in *ignore_set* before entering the CS (line 164).

We do not provide a proof of correctness for this version of the algorithm. Instead, in the next section we describe the enhancements necessary to make the algorithm work using only non-atomic reads and writes, and then prove the correctness of the enhanced algorithm. The same sequence of lemmas that are used in the proof of the enhanced algorithm could also be used to prove the correctness of the simpler algorithm in Figure 5.8, except the arguments used in the proofs of the lemmas could also be simplified in a number of places.

5.10 k-Exclusion With Ticket Resetting and FIFE (Non-atomic RWs)

In this section we enhance the algorithm in Figure 5.8 so that it works in the model in which we use only non-atomic reads and writes. The new algorithm is given in Figure 5.10 (with associated variable definitions in Figure 5.9). It satisfies k-exclusion, starvation freedom, FIFE, and resets tickets in the exit protocol. The doorway consists of lines 171..181, the waiting room consists of lines 182..196, and the exit protocol consists of lines 198..200.

We explain how we arrived at the algorithm in Figure 5.10 by explaining what can go wrong when using non-atomic reads and writes in the algorithm in Figure 5.8. One change we made has to do with the bypass mechanism at line 167 (lines 199..200 in the new algorithm). We reintroduced the condition check at line 200. This condition was present in the algorithm in Figure 5.7 (line 141), where we made use of non-atomic reads and writes, but was removed in the algorithm in Figure 5.8 (line 167), where we made use of atomic reads and writes. As we explained in Section 5.8, this condition is necessary to ensure that starvation freedom is not violated when non-atomic reads and writes are being used.

Another change, for a similar reason, affects the *DWYBypass* shared variable. Instead of blind-setting $DWYBypass[p][i]$ to true, as in line 150 in Figure 5.8, p now sets $DWYBypass[p][i]$ to true only if it previously found it to be false (see line 178 in Figure 5.10). Before a process p finishes the doorway, p needs to check whether there was some process q that executed the "main part" of the doorway concurrently with p. If this is the case, then p adds q to $p.ignore\_set$, and then does not attempt to capture q before entering the CS. To check if q executed the doorway concurrently with p, p checks at line 153 whether $Doorway[q][p] \vee DWYBypass[q][p]$ is true. The problem that can arise when using non-atomic reads and writes is that q may execute multiple passages after p executes line 146 and before p reaches line 153. In this case, p should read true from $DWYBypass[q][p]$ and add q to $p.ignore\_set$. However, if q writes $DWYBypass[q][p]$ at line 150 concurrently with p's read of $DWYBypass[q][p]$, p's read may incorrectly return false. To prevent this in the new algorithm, q writes $DWYBypass[q][p]$ at line 178 only if $DWYBypass[q][p]$ is false.

A similar problem can arise if q executes a passage multiple times and tries capturing p in each passage. In particular, q may repeatedly write $Capture[q][p]$ at line 164 at the same time as p reads $Capture[q][p]$ at line 163. This can result in p reading a value from $Capture[q][p]$ that is much smaller than the value that q writes, and p incorrectly

concluding that it has not been captured. As a result, FIFE may be violated. To fix this problem, we modify the algorithm so that q writes $Capture[q][p]$ at line 164 only if $Capture[q][p] < Ticket[p][q]$, i.e., q writes $Capture[q][p]$ only if the value in $Capture[q][p]$ is too small to allow p to detect that it has been captured. This is essentially what is being tested for in the new algorithm at line 195. The condition at line 195, however, is slightly more complex than we just described, due to there being three copies of $Capture[q][p]$ (i.e., $Capture[q][p][1]$, $Capture[q][p][2]$, and $Capture[q][p][3]$). The reason we need to use three copies of $Capture[q][p]$ is to protect against other bad scenarios, which we now describe.

If we only had one copy of $Capture[q][p]$ (i.e., $Capture[q][p][1]$), and references to the other two copies (i.e., $Capture[q][p][2]$, and $Capture[q][p][3]$) were deleted, then the following scenario could happen. The scenario is for $k = 2$ and involves processes p, q, and r.

1. Process p finishes the doorway before q starts the doorway.

2. Process q leaves the NCS, executes through the waiting room, and temporarily stops taking steps before attempting to capture p. That is, q temporarily stops taking steps before writing $Capture[q][p][1]$ (line 196).

3. Process p executes through the CS and returns to the NCS.

4. Process r leaves the NCS and enters the CS.

5. Process p leaves the NCS and enters the waiting room. It starts reading $Capture[q][p][1]$ to determine if it has been captured (line 193).

6. Process q starts writing $Capture[q][p][1]$ (line 196).

7. Process p incorrectly reads a value from $Capture[q][p][1]$ larger than its ticket, and detects that it has been captured.

8. Process p enters the CS.

9. Process q enters the CS.

10. Processes p, q, and r are all in the CS, and k-exclusion (for $k = 2$) is violated.

To fix this problem, we employ a more sophisticated version of a technique used in the algorithm in Figure 5.6 in Section 5.7. What we do is triplicate $Capture[q][p]$ (i.e., create $Capture[q][p][1]$, $Capture[q][p][2]$, and $Capture[q][p][3]$) and have process p read *and write* the copies at lines 176, 191..193 in the opposite order that q writes them at line 196. This prevents the above problematic scenario from happening. Process p may incorrectly read one of the copies of $Capture[q][p]$ due to a concurrent write by q. However, if p incorrectly reads more than one copy of $Capture[q][p]$ due to a concurrent write by q, it must be the case that q starts a subsequent passage and attempts to capture p a second time while p is in the waiting room. This is because p and q read/write the copies of $Capture[q][p]$ in the opposite order. In the above scenario, if q starts a second passage while p is in the waiting room, it will detect that p and r have smaller tickets, and thus will never advance to the part of the waiting room where q can attempt to capture p a second time.

In the algorithm in Figure 5.6, creating two copies of $Capture[q][p]$ was sufficient, but this is not the case in the algorithm in Figure 5.10. The latter algorithm requires a third copy of $Capture[q][p]$ because it resets *Capture* entries in the doorway, which is not done by the former algorithm. To understand what can go wrong in Figure 5.10 if we only had $Capture[q][p][1]$ and $Capture[q][p][2]$, suppose all references to $Capture[q][p][3]$ are removed, and consider the scenario below. The scenario is for $k = 2$, and involves processes p, q, and r (steps 1-4 are exactly as in the preceding scenario):

1. Process p finishes the doorway before q starts the doorway.

2. Process q leaves the NCS, executes through the waiting room, and temporarily

stops taking steps before attempting to capture p. That is, q temporarily stops taking steps before writing $Capture[q][p][1]$ (line 196).

3. Process p executes through the CS and returns to the NCS.

4. Process r leaves the NCS and enters the CS.

5. Process p leaves the NCS and enters the doorway. It sets $Capture[q][p][2] = 0$, and then starts writing 0 to $Capture[q][p][1]$.

6. Process q writes $Capture[q][p][1]$ concurrently with p's write. The end result of the concurrent writes is that $Capture[q][p][1]$ contains a value larger than the ticket that p chose.

7. Process q starts writing $Capture[q][p][2]$.

8. Process p reads $Capture[q][p][2]$ at line 192 concurrently with q's write, reading a value larger than the ticket p chose. Process p evaluates $ticket < Capture[q][p][2]$ to be true and proceeds to line 193.

9. Process p evaluates the condition $ticket < Capture[q][p][1]$ at line 193 to be true (see step 6 above), and sets $captured = \mathsf{true}$.

10. Process p enters the CS.

11. Process q enters the CS.

12. Processes p, q, and r are all in the CS, and k-exclusion (for $k = 2$) is violated.

Triplicating *Capture*[q][p] prevents the preceding scenario from happening. The problem was that p could read two "tainted" copies of *Capture*[q][p] in a row and erroneously detect that it was captured. It is impossible, however, for p to read three "tainted" copies of *Capture*[q][p] in a row and not actually have a clear path into the CS. If p reads "tainted" values in all three copies of *Capture*[q][p], either because of preceding concurrent writes, or a write concurrent with p's read, then q must have attempted to capture p twice, and in the second attempt q will have started its passage after p finished the doorway. In such a case, p can enter the CS without danger of breaking k-exclusion.

Note that the order in which the three disjunctions at line 195 are evaluated is unimportant for the correctness of the algorithm. Also, the variable *Ticket*[i][p] only needs to be read once per value of i at line 195, i.e., the same value of *Ticket*[i][p] can be used in the evaluation of each of the three disjunctions.

Figure 5.9: Variable definitions for Figure 5.10

shared variables:
Ticket: **array**[1..N][1..N] **of** $\mathbb{N}$ **init all 0**
Doorway: **array**[1..N][1..N] **of boolean init all** false
Bypass: **array**[1..N][1..N] **of boolean init all** false
Capture: **array**[1..N][1..N][1..3] **of** $\mathbb{N}$ **init all 0**
DWYBypass: **array**[1..N][1..N] **of boolean init all** false
(DSM model: *Ticket*[i][p], *Capture*[i][p][1..3], *Bypass*[i][p], *Doorway*[i][p] local to p $\forall i$)

private variables:
predecessor_set, *ignore_set*: **Set of** $\mathbb{N}$
ticket: $\mathbb{N}$
captured: **boolean**

Figure 5.10: (Non-atomic RWs) k-Exclusion with FIFE and ticket resetting for process $p \in \{1, ..., N\}$

```
169 loop
170     NCS
171     foreach i ∈ {1..N} do  Doorway[p][i] := true
172     foreach i ∈ {1..N} do  DWYBypass[i][p] := false
173     ticket := 1 + max(Ticket[1][p], Ticket[2][p], ..., Ticket[N][p])
174     foreach i ∈ {1..N} do  Ticket[p][i] := ticket
175     foreach i ∈ {1..N} do
176         Capture[i][p][3] := 0; Capture[i][p][2] := 0; Capture[i][p][1] := 0
177     foreach i ∈ {1..N} do
178         if ¬DWYBypass[p][i] then  DWYBypass[p][i] := true
179     ignore_set := ∅
180     foreach i ∈ {1..N} do
181         if Doorway[i][p] ∨ DWYBypass[i][p] then  ignore_set := ignore_set ∪ {i}
182     foreach i ∈ {1..N} do  Doorway[p][i] := false
183     foreach i ∈ {1..N} do  Bypass[i][p] := false
184     predecessor_set := {1..N} \ {p}
185     captured := false
186     while |predecessor_set| ≥ k ∧ ¬captured do
187         foreach i ∈ predecessor_set do
188             if Bypass[i][p] ∨ (¬Doorway[i][p] ∧ (Ticket[i][p] = 0 ∨ (ticket, p) <
                (Ticket[i][p], i))) then
189                 predecessor_set := predecessor_set \ {i}
190         foreach i ∈ {1..N} do
191             if ticket < Capture[i][p][3] then
192                 if ticket < Capture[i][p][2] then
193                     if ticket < Capture[i][p][1] then  captured := true
194     foreach i ∈ {1..N} \ ignore_set do
            // Evaluation order of disjunctions is unimportant.
195         if ⋁_{j∈{1..3}} Capture[p][i][j] ≤ Ticket[i][p] then
196             Capture[p][i][1] := ticket; Capture[p][i][2] := ticket; Capture[p][i][3] :=
                ticket
197     CS
198     foreach i ∈ {1..N} do  Ticket[p][i] := 0
199     foreach i ∈ {1..N} do
200         if ¬Bypass[p][i] then  Bypass[p][i] := true
201 end loop
```

One of the interesting aspects of this algorithm that was not part of Lamport's original

Bakery algorithm [32] is that it uses multi-writer multi-reader safe registers, i.e., multiple processes can write the same shared variable concurrently and the result of such a write is arbitrary. In particular, as demonstrated in the scenario above, processes p and q can concurrently write $Capture[q][p][*]$. Surprisingly, despite the fact that $Capture[q][p][*]$ can store an arbitrary value after a concurrent write, the algorithm is still correct. All prior work on mutual exclusion algorithms using safe registers that we are familiar with, including of course Lamport's Bakery algorithm, employs only single-writer multi-reader registers.

In the remainder of this section we prove the algorithm's correctness.

5.10.1 Preliminary Lemmas

Let C_p denote the operation in which process p chooses its ticket at line 173, let A_p denote the operation in which p announces its ticket at line 174, and let EP_p denote the operation in which p executes its exit protocol at lines 198..200.

The following lemma and corollary are analogues to Lemma 5.34 and Corollary 5.35. The proof of the lemma below is nearly the same as the proof of Lemma 5.34, modulo line number differences.

Lemma 5.48. *Let p and q be distinct processes. If $A_p \rightarrow C_q$ and $C_q \rightarrow EP_p$, then $t_p < t_q$, where t_p and t_q are the tickets chosen by p and q, respectively.*

Proof. Process p announces its ticket (line 174) before q chooses its ticket (line 173), and process q chooses its ticket (line 173) before p resets its ticket (line 198). Therefore the value that q reads from $Ticket[p][q]$ at line 173 is equal to t_p. Process q adds one to this value, and so the ticket t_q that q chooses satisfies $t_q > t_p$. □

As a corollary to the preceding lemma, we state it in the following logically equivalent form:

Corollary 5.49. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q. If $t_q \leq t_p$ and $C_q \rightarrow EP_p$, then $C_q \dashrightarrow A_p$.*

The following lemma is an analogue to Lemma 5.36. Its proof follows the same structure as the proof of Lemma 5.36.

Lemma 5.50. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q, respectively. If $(t_q, q) < (t_p, p)$ and $C_q \dashrightarrow A_p$ then p does not remove q from $p.predecessor\_set$ before EP_q starts.*

Proof. Assume that $(t_q, q) < (t_p, p)$ and that $C_q \dashrightarrow A_p$. Suppose, by way of contradiction, that p removes q from $p.predecessor\_set$ before EP_q starts.

Before p removes q from its predecessor set, p reads $Bypass[q][p]$, $Doorway[q][p]$, and $Ticket[q][p]$ (in that order) to evaluate the condition at line 188. Let R_p^B, R_p^D, and R_p^T be these reads, respectively.

Claim 5.50.1. *R_p^B returns false.*

Proof. The only processes that write into $Bypass[q][p]$ are p and q. Process p sets $Bypass[q][p] = \mathsf{false}$ at line 183 (denote this write operation W_p^B) and process q may set $Bypass[q][p] = \mathsf{true}$ at line 200 of the exit protocol. There are no other places where $Bypass[q][p]$ is written. By inspection of the algorithm, $W_p^B \rightarrow R_p^B$.

Let W_q^B be any write operation by q into $Bypass[q][p]$. It remains to show that either $W_q^B \rightarrow W_p^B$ or $R_p^B \rightarrow W_q^B$. (To see why this suffices, note that if this is the case for all writes W_q^B by q into $Bypass[q][p]$, there is a well-defined last write into $Bypass[q][p]$ that precedes R_p^B, namely W_p^B, which sets this variable false, and there are no writes into $Bypass[q][p]$ concurrent with R_p^B.)

As we observed, q only writes $Bypass[q][p]$ in its exit protocol. So, to prove that either $W_q^B \rightarrow W_p^B$ or $R_p^B \rightarrow W_q^B$ it suffices to prove that: (1) if EP'_q is q's execution of the exit protocol in its preceding passage, then $EP'_q \rightarrow W_p^B$, and (2) $R_p^B \rightarrow EP_q$.

For (1) we have that, by the structure of the algorithm and the hypothesis of the lemma, $EP'_q \rightarrow C_q \dashrightarrow A_p \rightarrow W_p^B$, from which it follows that $EP'_q \rightarrow W_p^B$.

For (2) we have that, by the structure of the algorithm, R_p^B ends before p removes q from *p.predecessor_set*, which, by the hypothesis of this lemma, happens before EP_q starts. Thus, $R_q^B \rightarrow EP_q$. (Claim 5.50.1) □

By Claim 5.50.1, and lines 188..189, for p to remove q from *p.predecessor_set*, R_p^D must return false, and R_p^T must return 0 or some value t such that $(t_p, p) < (t, q)$.

Let W_q^D and $W_q^{D^+}$ be the successive write operations in which q writes, respectively, true and false into *Doorway*$[q][p]$ at lines 171 and 182.

Claim 5.50.2. $W_q^{D^+} \dashrightarrow R_p^D$

Proof. By assumption and the structure of the algorithm, $W_q^D \rightarrow C_q \dashrightarrow A_p \rightarrow R_p^D$, and so $W_q^D \rightarrow R_p^D$. If $R_p^D \rightarrow W_q^{D^+}$, then $W_q^D \rightarrow R_p^D \rightarrow W_q^{D^+}$. That is, R_p^D occurs entirely between two successive writes to *Doorway*$[q][p]$, and hence must return the value written by W_q^D, which is true. This, however, contradicts that R_p^D must return false. Therefore it is not the case that $R_p^D \rightarrow W_q^{D^+}$, and so, $W_q^{D^+} \dashrightarrow R_p^D$. (Claim 5.50.2) □

Let W_q^T and $W_q^{T^+}$ be the successive write operations in which q writes t_q and 0, respectively, into *Ticket*$[q][p]$ at lines 174 and 198. By inspection of the order of operations on line 188, we have that $R_p^D \rightarrow R_p^T$. This and Claim 5.50.2 imply that $W_q^T \rightarrow W_q^{D^+} \dashrightarrow R_p^D \rightarrow R_p^T$, and so $W_q^T \rightarrow R_p^T$.

R_p^T finishes before process p removes q from *p.predecessor_set*, and by assumption, process p removes q from *p.predecessor_set* before EP_q starts. Therefore $R_p^T \rightarrow EP_q$. Furthermore, $W_q^{T^+} \subseteq EP_q$, and so $R_p^T \rightarrow W_q^{T^+}$. This and the fact that $W_q^T \rightarrow R_p^T$ (established in the preceding paragraph) imply that $W_q^T \rightarrow R_p^T \rightarrow W_q^{T^+}$. That is, R_p^T occurs between two successive writes into *Ticket*$[q][p]$, and hence must return the value written by W_q^T, which is t_q. Processes always choose non-zero tickets, so $t_q \neq 0$.

Recall that R_p^B, R_p^D, and R_p^T are p's reads of $Bypass[q][p]$, $Doorway[q][p]$, and $Ticket[q][p]$ when p evaluates the condition at line 188 before p removes q from $p.predecessor\_set$. By Claim 5.50.1, R_p^B returns false. So, by the condition at line 188, R_p^D must return false, and R_p^T must return zero or a value t such that $(t_p, p) < (t, q)$. We established in the preceding paragraph that R_p^T returns $t_q \neq 0$. Therefore $(t_p, p) < (t_q, q)$, contrary to the assumption that $(t_q, q) < (t_p, p)$. □

5.10.2 Capturing Lemmas

We now present some lemmas about the capturing mechanism. These lemmas will be used to prove that k-exclusion holds.

We previously defined what it means for a process to attempt to capture another process, and also what it means for a process to detect capture, in Section 5.6 and Section 5.7. As a result of the triplication of the entries in the *Capture* array, we need to redefine these terms here.

Let D_q^p be the operation in which a process q *detects capture* by a process p at lines 191–193, i.e., D_q^p is the operation in which q reads $q.ticket < Capture[p][q][3]$, $q.ticket < Capture[p][q][2]$, and $q.ticket < Capture[p][q][1]$ and consequently sets $q.captured = \mathsf{true}$ at line 193. Let X_p^q denote the operation in which a process p *attempts to capture* a process q at line 196, i.e., X_p^q is the operation in which p writes its ticket to $Capture[q][p][1]$, $Capture[q][p][2]$, and $Capture[q][p][3]$ at line 196. Note that these operations are non-atomic.

Recall the discussion in Section 5.9 where we illustrated that processes should not attempt to capture each other if the passage of the "main" part of their doorways is concurrent. The reason for this is that if processes do attempt to capture each other in an unrestricted manner, a situation can arise in which k-exclusion is violated. The following lemma — more precisely, its contrapositive, stated in Corollary 5.52 — essentially says that if a process tries capturing another one, then the processes did not execute the

"main" part of their doorways concurrently. This lemma is "new" in that it has no analogue in any of the preceding sections.

Lemma 5.51. *If some processes r and p execute passages such that C_r --↠ A_p and C_p --↠ A_r (i.e., r and p execute lines 173..174 concurrently), then r does not attempt to capture p.*

Proof. Assume that some processes r and p execute passages such that C_r --↠ A_p and C_p --↠ A_r. Either r reads true in $Doorway[p][r]$ at line 181, or not. In the former case, r adds p to $r.ignore\_set$ at line 181. By inspection of line 194, r does not attempt to capture any process in $r.ignore\_set$, and so the lemma holds.

In the latter case r reads false in $Doorway[p][r]$ at line 181. Let R_r^D denote this read, and let R_r^B denote r's read of $DWYBypass[p][r]$ at line 181. We argue that R_r^B returns true.

Let W_r^B be the write by r at line 172 where r sets $DWYBypass[p][r] = \mathsf{false}$, and let R_p^B be p's read of $DWYBypass[p][r]$ at line 178. Process p may write true to $DWYBypass[p][r]$ at line 178 depending on the value returned by R_p^B. If the write occurs, let W_p^B denote the write operation in which p writes true to $DWYBypass[p][r]$ at line 178. If the write does not occur, then W_p^B denotes the first read operation that p executes after R_p^B finishes (either the read of another $DWYBypass$ entry in the next iteration of the loop at line 177, or the read of a $Doorway$ entry in the first iteration of the loop at line 181).

Claim 5.51.1. $W_r^B \to R_p^B \to W_p^B \to R_r^B$

Proof. By assumption, C_r --↠ A_p, and so, by the structure of the algorithm: $W_r^B \to C_r$ --↠ $A_p \to R_p^B \to W_p^B$. This implies that $W_r^B \to R_p^B \to W_p^B$. It remains to show that $W_p^B \to R_r^B$. Suppose, by way of contradiction, that this is not the case, i.e., R_r^B --↠ W_p^B.

Let W_p^D and $W_p^{D^+}$ be p's successive writes of true and false to $Doorway[p][r]$ at line 171 and line 182, respectively. Recall that R_r^D is r's read of $Doorway[p][r]$ at line 181 and returns false. By assumption, C_p --↠ A_r, and so: $W_p^D \to C_p$ --↠ $A_r \to R_r^D \to R_r^B$ --↠

$W_p^B \to W_p^{D^+}$. This implies that $W_p^D \to R_r^D \to W_p^{D^+}$. W_p^D and $W_p^{D^+}$ are successive writes to $Doorway[p][r]$, so R_r^D must return the value written by W_p^D, which is true. However, this contradicts that R_r^D returns false. (Claim 5.51.1) □

We will now prove that R_r^B returns true. There are two cases, depending on the value returned by R_p^B.

Case 1: R_p^B returns false. In this case, W_p^B is an operation that writes true to $DWYBypass[p][r]$. Let W be any write operation on $DWYBypass[p][r]$ other than W_p^B. We will prove that either $W \to W_p^B$ or $R_r^B \to W$. Once we do so, since $W_p^B \to R_r^B$ (by Claim 5.51.1), it follows that R_r^B returns the same value as written by W_p^B, i.e., true, as wanted. So it remains to prove that for any write operation W on $DWYBypass[p][r]$ other than W_p^B, either $W \to W_p^B$ or $R_r^B \to W$. By inspection of the algorithm, only p and r ever write $DWYBypass[p][r]$, at line 178 and line 172, respectively. So, W is either performed by process p or process r:

Case 1a: W is performed by process r. Observing that process r only writes $DWYBypass[p][r]$ at line 172, and all writes by r at this line are totally ordered by $\to$, Claim 5.51.1 implies that $W \to W_p^B$ (if W is W_r^B or an earlier write) or $R_r^B \to W$ (if W occurs in a later passage by r than the one in which W_r^B and R_r^B occur), as wanted. **(End of Case 1a)**

Case 1b: W is performed by process p. If $W \to W_p^B$ then we are done. Otherwise, $W_p^B \to W$, since (i) W is not W_p^B (by assumption), and (ii) all writes by p to $DWYBypass[p][r]$ are totally ordered by $\to$. In this case we will prove that $R_r^B \to W$. In fact, it suffices to prove that $R_r^B \to W$ for the first (in the sense of the total order defined by $\to$) write operation W on $DWYBypass[p][r]$ by process p such that $W_p^B \to W$. (This is well-defined, since $\to$ totally orders all operations of p.) To see this, let R be the read operation on $DWYBypass[p][r]$ (at line 178) by process p that returns false and immediately precedes W.

Claim 5.51.2. *There exists a write W_r^{B+} by process r on $DWYBypass[p][r]$ (at line 172) such that (i) $W_r^B \rightarrow W_r^{B+}$, and (ii) $W_r^{B+} \dashrightarrow R$.*

Proof. Suppose, by way of contradiction, that the claim is false. That is, assume that either W_r^B is the last operation by r to write to $DWYBypass[p][r]$, or for every write $\hat{W}$ by process r on $DWYBypass[p][r]$ if $W_r^B \rightarrow \hat{W}$ then $R \rightarrow \hat{W}$.

Since R and W belong to a later passage of p than R_p^B and W_p^B, we have $W_p^B \rightarrow R$. Also, by Claim 5.51.1, $W_r^B \rightarrow W_p^B$. So, we have that $W_r^B \rightarrow W_p^B \rightarrow R$. For every write $\hat{W}$ on $DWYBypass[p][r]$ by r, either $\hat{W} \rightarrow W_p^B$ (if $\hat{W} = W_r^B$, or is a preceding write) or $R \rightarrow \hat{W}$ (if W_r^B is not the last write by r to $DWYBypass[p][r]$). Similarly, for every write $\hat{W}$ on $DWYBypass[p][r]$ by p other than W_p^B, either $\hat{W} \rightarrow W_p^B$ or $R \rightarrow \hat{W}$ (if $\hat{W} = W$ or a later write). Since no process other than p or r writes $DWYBypass[p][r]$, it follows that R returns the value written by W_p^B, i.e., true. This contradicts that R returns false. □

Consider the write W_r^{B+} that exists by Claim 5.51.2. Since $W_r^B \rightarrow W_r^{B+}$, W_r^{B+} must occur in a passage by r after the one in which r executes W_r^B and R_r^B. Thus $R_r^B \rightarrow W_r^{B+} \dashrightarrow R \rightarrow W$, and so $R_r^B \rightarrow W$, as we wanted for this subcase. **(End of Case 1b, and Case 1.)**

Case 2: R_p^B returns true. In this case, W_p^B is **not** a write operation on $DWYBypass[p][r]$ — it is merely an operation that reads some other variable. (In particular, by definition, W_p^B is either the read by p of another *DWYBypass* entry in the next iteration of the loop at line 177 after R_p^B, or the read of a *Doorway* entry by p in the first iteration of the loop at line 181). By Claim 5.51.1, $R_p^B \rightarrow R_r^B$. We will now prove that for every write W on $DWYBypass[p][r]$, either $W \rightarrow R_p^B$, or $R_r^B \rightarrow W$. Once we do so, it follows that R_p^B and R_r^B return the same value; since, by the hypothesis of this case, R_p^B returns true, so does R_r^B, as wanted.

It remains to prove that for every write W on $DWYBypass[p][r]$, either $W \rightarrow R_p^B$, or $R_r^B \rightarrow W$. By inspection of the algorithm, only p and r ever write $DWYBypass[p][r]$, at

line 178 and line 172, respectively. So, W is either performed by process p or process r:

Case 2a: W is performed by process r. Observing that process r only writes $DWYBypass[p][r]$ at line 172, and all writes by r at this line are totally ordered by $\rightarrow$, Claim 5.51.1 implies that $W \rightarrow R_p^B$ (if $W = W_r^B$ or belongs to an earlier passage), or $R_r^B \rightarrow W$ (if W belongs to a later passage of r), as wanted. **(End of Case 2a)**

Case 2b: W is performed by process p. If $W \rightarrow R_p^B$ then we are done. Otherwise, $R_p^B \rightarrow W$, since all operations by p are totally ordered by $\rightarrow$. In this case we will prove that $R_r^B \rightarrow W$. In fact, it suffices to prove that $R_r^B \rightarrow W$ for the first (in the sense of the total order defined by $\rightarrow$) write operation W on $DWYBypass[p][r]$ by process p such that $R_p^B \rightarrow W$. (This is well-defined, since $\rightarrow$ totally orders all operations of p.) To see this, let R be the read operation on $DWYBypass[p][r]$ (at line 178) by process p that returns false and immediately precedes W. Note that R and W belong to a later passage by p than R_p^B and W_p^B.

(The following claim, although having an identical statement to Claim 5.51.2 proven in Case 1b, must be proven again for the assumptions made in this case. In particular, Claim 5.51.2 uses the fact the W_p^B writes true, whereas in this case W_p^B is not a write operation. The claim below uses the fact that R_p^B returns true.)

Claim 5.51.3. *There exists a write W_r^{B+} by process r on $DWYBypass[p][r]$ (at line 172) such that (i) $W_r^B \rightarrow W_r^{B+}$, and (ii) $W_r^{B+} \dashrightarrow R$.*

Proof. Suppose, by way of contradiction, that the claim is false. That is, assume that either W_r^B is the last operation by r to write to $DWYBypass[p][r]$, or for every write $\hat{W}$ by process r on $DWYBypass[p][r]$ if $W_r^B \rightarrow \hat{W}$ then $R \rightarrow \hat{W}$.

Recall, $R_p^B \rightarrow W$, R and W occur in the same passage, and W does not occur in the same passage of p as R_p^B. This implies that $R_p^B \rightarrow R$. We will now prove that for every write $\hat{W}$ on $DWYBypass[p][r]$ either $\hat{W} \rightarrow R_p^B$, or $R \rightarrow \hat{W}$. This implies that R_p^B and R return the same value. This is a contradiction, since R_p^B returns true by the

assumption of Case 2, and R returns false. So it remains to prove that for every write $\hat{W}$ on $DWYBypass[p][r]$ either $\hat{W} \rightarrow R_p^B$, or $R \rightarrow \hat{W}$.

Only process p and r write $DWYBypass[p][r]$. First, consider writes by process r. By Claim 5.51.1, $W_r^B \rightarrow R_p^B$. So, for every write $\hat{W}$ on $DWYBypass[p][r]$ by r, either $\hat{W} \rightarrow R_p^B$ (if $\hat{W} = W_r^B$, or $\hat{W} \rightarrow W_r^B$) or $R \rightarrow \hat{W}$ (if $W_r^B \rightarrow \hat{W}$). Now consider writes by process p. Recall that p does not write $DWYBypass[p][r]$ in the passage to which R_p^B belongs, that W is the first write to $DWYBypass[p][r]$ after R_p^B, and $R \rightarrow W$. Thus, for every write $\hat{W}$ on $DWYBypass[p][r]$ by p, either $\hat{W} \rightarrow R_p^B$ (if $\hat{W}$ occurs in a passage prior to R_p^B) or $R \rightarrow \hat{W}$ (if $\hat{W} = W$ or a later write). □

Consider the write W_r^{B+} that exists by Claim 5.51.3. By the structure of the algorithm, and $W_r^B \rightarrow W_r^{B+}$, it must be the case that W_r^{B+} occurs in a passage by r after the one in which r executes W_r^B and R_r^B. Thus $R_r^B \rightarrow W_r^{B+} \dashrightarrow R \rightarrow W$, and so $R_r^B \rightarrow W$, as we wanted for this subcase. **(End of Case 2b, and Case 2.)**

The preceding case analysis establishes that R_r^B must return true, and so r will add p to $r.ignore\_set$. This means that r will not attempt to capture p, as explained earlier for the case when r reads true in $Doorway[p][r]$ at line 181. □

As a corollary to the preceding lemma, we state its contrapositive.

Corollary 5.52. *If a process r attempts to capture a process p, then every passage by p is such that either $A_p \rightarrow C_r$ or $A_r \rightarrow C_p$ holds.*

The following lemma relates capture detection and capture attempts by showing that when a process detects capture, there must exist some other process that previously attempted to capture it. The lemma is an analogue to Lemma 5.24, but its proof is more complex since it needs to take into account that capture attempts happen in a more restricted manner than in previous algorithms, *Capture* entries are triplicated, and *Capture* entries are reset to 0 in the doorway.

Lemma 5.53. *Suppose a process p detects capture by a process r in a passage in which it chooses a ticket t_p. Then there exists a passage by r in which the following are true: (i) $A_p \rightarrow C_r$, (ii) r chooses a ticket $t_r > t_p$, and (iii) $X_r^p \dashrightarrow D_p^r$.*

Proof. Process p detects capture by a process r in a passage in which it chooses a ticket $t_p > 0$. This implies that p reads values $v_{r3} > t_p > 0$, $v_{r2} > t_p > 0$, and $v_{r1} > t_p > 0$ from *Capture*$[r][p][3]$, *Capture*$[r][p][2]$, and *Capture*$[r][p][1]$ at lines 191, 192, 193, respectively. We use $R_{p3} \subseteq D_p^r$, $R_{p2} \subseteq D_p^r$, and $R_{p1} \subseteq D_p^r$ to denote these read operations. Process p only ever writes 0 to *Capture*$[r][p][3]$, *Capture*$[r][p][2]$, and *Capture*$[r][p][1]$ at line 176, and so some process other than p must write *Capture*$[r][p][3]$, *Capture*$[r][p][2]$, and *Capture*$[r][p][1]$ before D_p^r ends. The only process other than p that writes *Capture*$[r][p][1]$, *Capture*$[r][p][2]$ and *Capture*$[r][p][3]$ is process r, at line 196, where r attempts to capture p by writing its ticket to these variables. Thus there exists a passage by r in which r starts an attempt to capture p before D_p^r ends, i.e., $X_r^p \dashrightarrow D_p^r$. Consider the last such passage by r that starts prior to the end of D_p^r. We first show that $A_p \rightarrow C_r$, and then conclude by explaining why the ticket t_r that r chooses in this passage satisfies $t_r > t_p$.

Suppose, by way of contradiction, that $A_p \rightarrow C_r$ is false. Let W_{r1}, W_{r2}, and W_{r3} be the write operations (part of X_r^p) in which r writes its ticket t_r to *Capture*$[r][p][1]$, *Capture*$[r][p][2]$, and *Capture*$[r][p][3]$, respectively, at line 196. Also, let W_{p3}, W_{p2}, and W_{p1} denote the writes by p in which p sets *Capture*$[r][p][3] = 0$, *Capture*$[r][p][2] = 0$, and *Capture*$[r][p][1] = 0$ at line 176.

Claim 5.53.1. $C_p \rightarrow EP_r$

Proof. Suppose, by way of contradiction, that $EP_r \dashrightarrow C_p$. Thus $W_{r3} \rightarrow EP_r \dashrightarrow C_p \rightarrow W_{p3} \rightarrow R_{p3}$, and so $W_{r3} \rightarrow W_{p3} \rightarrow R_{p3}$. We now show that R_{p3} reads the value written by W_{p3}, i.e., 0, contradicting that R_{p3} returns a value $v_{r3} > t_p > 0$.

Since $W_{p3} \rightarrow R_{p3}$, we can show that R_{p3} reads the value written by W_{p3} by showing

that for any write $\hat{W}$ to *Capture*[r][p][3] other than W_{p3}, either $\hat{W} \to W_{p3}$ or $R_{p3} \to \hat{W}$. Process p and r are the only processes that write *Capture*[r][p][3]. First, consider writes by process r, which occur only once per passage at line 196 when r attempts to capture p. We know that $W_{r3} \to W_{p3}$ (established in the last paragraph), and that W_{r3} is part of the last attempt by r to capture p that starts before the end of D_p^r. Thus, for every write $\hat{W}$ by r to *Capture*[r][p][3], either $\hat{W} \to W_{p3}$ (if $\hat{W}$ is W_{r3} or an earlier write) or $D_p^r \to \hat{W}$ (if $\hat{W}$ is part of an attempt by r to capture p that starts after the end of D_p^r). Since $R_{p3} \subseteq D_p^r$, either $\hat{W} \to W_{p3}$ or $R_{p3} \to \hat{W}$, as desired. Next, consider writes to *Capture*[r][p][3] by process p. Process p only writes *Capture*[r][p][3] once per passage at line 176, and W_{p3} and R_{p3} are part of the same passage. Hence, for every write $\hat{W}$ by p to *Capture*[r][p][3] other than W_{p3}, either $\hat{W} \to W_{p3}$ (if $\hat{W}$ occurs in an earlier passage than W_{p3} and R_{p3}) or $R_{p3} \to \hat{W}$ (if $\hat{W}$ occurs in a later passage than W_{p3} and R_{p3}), as desired.

Thus R_{p3} reads the value written by W_{p3}, i.e., 0, contradicting that R_{p3} returns a value $v_{r3} > t_p > 0$. (Claim 5.53.1) □

Claim 5.53.2. *The ticket t_r that r chooses (and writes in W_{r1}, W_{r2}, and W_{r3}) satisfies $t_r < t_p$.*

Proof. By the supposition that $A_p \to C_r$ is false, and Corollary 5.52, $A_r \to C_p$. This, Claim 5.53.1, and Lemma 5.48 imply that the ticket t_r that r chooses satisfies $t_r < t_p$. (Claim 5.53.2) □

Either $W_{r2} \to R_{p2}$ or $R_{p2} \dashrightarrow W_{r2}$:

Case 1: $W_{r2} \to R_{p2}$**.** Either $W_{p2} \to W_{r2}$ or $W_{r2} \dashrightarrow W_{p2}$:

Subcase 1a: $W_{p2} \to W_{r2}$**.** By the assumption of this case, $W_{p2} \to W_{r2} \to R_{p2}$. We will now show that the value v_{r2} returned by R_{p2} is the same as the value written by W_{r2}, i.e., $v_{r2} = t_r$. By Claim 5.53.2, the value t_r written by W_{r2} satisfies $t_r < t_p$, and so

$v_{r2} < t_p$. This contradicts that $v_{r2} > t_p$. It remains to show that the value returned by R_{p2} is the same as the value written by W_{r2}.

Since $W_{r2} \rightarrow R_{p2}$, we can show that R_{p2} reads the value written by W_{r2} by showing that for any write $\hat{W}$ to *Capture*$[r][p][2]$ other than W_{r2}, either $\hat{W} \rightarrow W_{r2}$ or $R_{p2} \rightarrow \hat{W}$. Process p and r are the only processes that write *Capture*$[r][p][2]$. First, consider writes by process p, which occur only once per passage at line 176. Since $W_{p2} \rightarrow W_{r2}$, and W_{p2} and R_{p2} are part of the same passage by p, it follows that for every write $\hat{W}$ by p to *Capture*$[r][p][2]$, either $\hat{W} \rightarrow W_{r2}$ (if $\hat{W}$ is W_{p2} or an earlier write) or $R_{p2} \rightarrow \hat{W}$ (if $\hat{W}$ occurs in a passage by p later than R_{p2}), as desired. Next, consider writes by process r, which occur only once per passage at line 196 when r attempts to capture p. We know that W_{r2} is part of the last attempt by r to capture p that starts before the end of D_p^r. Thus, for any write $\hat{W}$ by r to *Capture*$[r][p][2]$ other than W_{r2}, either $\hat{W} \rightarrow W_{r2}$ (if $\hat{W}$ occurs in a passage earlier than W_{r2}) or $D_p^r \rightarrow \hat{W}$ (if $\hat{W}$ is part of an attempt by r to capture p that starts after the end of D_p^r). Since $R_{p2} \subseteq D_p^r$, either $\hat{W} \rightarrow W_{r2}$ or $R_{p2} \rightarrow \hat{W}$, as desired.

Subcase 1b: $W_{r2} \dashrightarrow W_{p2}$**.** In this case, $W_{r1} \rightarrow W_{r2} \dashrightarrow W_{p2} \rightarrow W_{p1} \rightarrow R_{p1}$, and so $W_{r1} \rightarrow W_{p1} \rightarrow R_{p1}$. We will now show that the value v_{r1} returned by R_{p1} is the same value written by W_{p1}, i.e., 0, which contradicts that R_{p1} returns a value $v_{r1} > t_p > 0$.

Since $W_{p1} \rightarrow R_{p1}$, we can show that R_{p1} reads the value written by W_{p1} by showing that for any write $\hat{W}$ to *Capture*$[r][p][1]$ other than W_{p1}, either $\hat{W} \rightarrow W_{p1}$ or $R_{p1} \rightarrow \hat{W}$. Process p and r are the only processes that write *Capture*$[r][p][1]$. First, consider writes by process r, which occur only once per passage at line 196 when r attempts to capture p. $W_{r1} \rightarrow W_{p1}$ and W_{r1} is part of the last attempt by r to capture p that starts before the end of D_p^r. Thus, for any write $\hat{W}$ by process r to *Capture*$[r][p][1]$, either $\hat{W} \rightarrow W_{p1}$ (if $\hat{W}$ is W_{r1} or an earlier write) or $D_p^r \rightarrow \hat{W}$ (if $\hat{W}$ is part of an attempt by r to capture p that starts after the end of D_p^r). Since $R_{p1} \subseteq D_p^r$, either $\hat{W} \rightarrow W_{p1}$ or $R_{p1} \rightarrow \hat{W}$, as desired. Next, consider writes by process p. Process p only writes *Capture*$[r][p][1]$ once

per passage at line 176, and W_{p1} and R_{p1} are part of the same passage by p. Thus, for any write $\hat{W}$ by p to *Capture*$[r][p][1]$ other than W_{p1}, either $\hat{W} \rightarrow W_{p1}$ or $R_{p1} \rightarrow \hat{W}$, as desired.

Case 2: $R_{p2} \dashrightarrow W_{r2}$**.** In this case, $W_{p3} \rightarrow R_{p3} \rightarrow R_{p2} \dashrightarrow W_{r2} \rightarrow W_{r3}$, and so $W_{p3} \rightarrow R_{p3} \rightarrow W_{r3}$.

Let W'_{r3} be r's last write to *Capture*$[r][p][3]$ before W_{r3} starts. (W'_{r3} must exist, otherwise R_{p3} returns the value written by W_{p3}, i.e., 0, contrary to the fact that R_{p3} returns a value $v_{r3} > t_p > 0$.) Either $W'_{r3} \rightarrow W_{p3}$ or $W_{p3} \dashrightarrow W'_{r3}$.

Subcase 2a: $W'_{r3} \rightarrow W_{p3}$**.** In this case, $W'_{r3} \rightarrow W_{p3} \rightarrow R_{p3} \rightarrow W_{r3}$. We will now show that R_{p3} returns that same value written by W_{p3}, i.e., 0, which contradicts that R_{p3} returns a value $v_{r3} > t_p > 0$.

Since $W_{p3} \rightarrow R_{p3}$, we can show that R_{p3} reads the value written by W_{p3} by showing that for any write $\hat{W}$ to *Capture*$[r][p][3]$ other than W_{p3}, either $\hat{W} \rightarrow W_{p3}$ or $R_{p3} \rightarrow \hat{W}$. Process p and r are the only processes that write *Capture*$[r][p][3]$. First, consider writes by process r. W'_{r3} and W_{r3} are successive writes by process r to *Capture*$[r][p][3]$, and $W'_{r3} \rightarrow W_{p3} \rightarrow R_{p3} \rightarrow W_{r3}$. So for any write $\hat{W}$ by process r to *Capture*$[r][p][3]$, either $\hat{W} \rightarrow W_{p3}$ (if $\hat{W}$ is W'_{r3} or an earlier write) or $R_{p3} \rightarrow \hat{W}$ (if $\hat{W}$ is W_{r3} or a later write), as desired. Next, consider writes by process p. Process p only writes *Capture*$[r][p][3]$ once per passage at line 176, and W_{p3} and R_{p3} are part of the same passage by p. Thus for any write $\hat{W}$ by p to *Capture*$[r][p][3]$ other than W_{p3}, either $\hat{W} \rightarrow W_{p3}$ or $R_{p3} \rightarrow \hat{W}$, as desired.

Subcase 2b: $W_{p3} \dashrightarrow W'_{r3}$**.** In this case $A_p \rightarrow W_{p3} \dashrightarrow W'_{r3} \rightarrow C_r$, and so $A_p \rightarrow C_r$. This contradicts our supposition that $A_p \rightarrow C_r$ is false.

Each part of the preceding case analysis yields a contradiction, and so $A_p \rightarrow C_r$ must be true.

Finally, we conclude by explaining why $t_p < t_r$: Process r starts its attempt to capture p before p detects capture, i.e., $X_r^p \dashrightarrow D_p^r$, and so, by the structure of the algorithm,

$C_r \rightarrow X_r^p \dashrightarrow D_p^r \rightarrow EP_p$. This implies that $C_r \rightarrow EP_p$, and therefore, by Lemma 5.48, $t_p < t_r$.

□

5.10.3 Main Proofs

We are now ready to prove the correctness of the algorithm. We first prove that the algorithm satisfies k-exclusion. In proving k-exclusion, we first provide a more technical lemma, Lemma 5.54, and then use this lemma to prove k-exclusion in Lemma 5.55. After this, we prove that FIFE holds (Lemma 5.57). The capturing mechanism added to the FIFE algorithm in Figure 5.10 introduces the possibility that a process can enter the CS via a new path, but it does not create any new obstacles that would prevent a process from entering the CS. Thus the proof of starvation freedom remains identical to the one given for the algorithm in Figure 5.7 (Lemma 5.41), modulo line number changes. Finally, we state and prove the RMR complexity of this algorithm (Lemma 5.59).

Let Q_p denote the first read operation in a passage that p executes after advancing past the loop at line 186. After Q_p finishes, we say that p is *CS-qualified*.

The following lemma is an analogue to Lemma 5.25, and is critical for proving k-exclusion. Its proof turns out to be more complex than the proof of Lemma 5.25, since it needs to take into account ticket resetting.

Lemma 5.54. *Let Y be a set of processes such that $|Y| \geq k$, and let p be a process that chooses a ticket t_p such that for all $i \in Y$, $(t_p, p) > (t_i, i)$, where t_i is the ticket chosen by i. If $\forall i \in Y, C_i \dashrightarrow A_p$ then p is not CS-qualified until some process in Y starts the exit protocol.*

Proof. Assume $\forall i \in Y, C_i \dashrightarrow A_p$. Suppose, by way of contradiction, that p is CS-qualified before any process in Y starts the exit protocol, i.e., Q_p finishes before any process in Y starts the exit protocol. Further assume that the lemma is not violated

prior to the start of Q_p. (If the lemma is violated earlier, then we can modify our choice of process p and its passage so that the preceding assumption holds.) By Lemma 5.50, p does not remove from *p.predecessor_set* any process in Y until some process in Y starts the exit protocol. This and the fact that $|Y| \geq k$ means that p cannot advance past the loop at line 186, and hence start Q_p, until either p sets *captured* = true or some process in Y starts the exit protocol. By assumption, Q_p finishes before any process in Y starts the exit protocol, and so p starts Q_p after p sets *captured* = true, i.e., Q_p starts after p detects capture by some process r. This and Lemma 5.53 imply that there exists a passage by r in which the following are true: (i) $A_p \rightarrow C_r$, (ii) r chooses a ticket $t_r > t_p$, and (iii) $X_r^p \dashrightarrow D_p^r$.

Claim 5.54.1. *For all $i \in Y$, $(t_r, r) > (t_i, i)$, where t_i is the ticket chosen by i.*

Proof. By part (ii) of Lemma 5.53, $(t_r, r) > (t_p, p)$. This and the assumption that for all $i \in Y$, $(t_p, p) > (t_i, i)$, imply that for all $i \in Y$, $(t_r, r) > (t_i, i)$. (Claim 5.54.1) □

Claim 5.54.2. $\forall i \in Y, C_i \dashrightarrow A_r$

Proof. By the hypothesis of the lemma, part (i) of Lemma 5.53, and the structure of the algorithm we have that for all $i \in Y, C_i \dashrightarrow A_p \rightarrow C_r \rightarrow A_r$, and therefore $C_i \dashrightarrow A_r$. (Claim 5.54.2) □

Claim 5.54.3. $Q_r \rightarrow Q_p$.

Proof. By the structure of the algorithm and part (iii) of Lemma 5.53, $Q_r \rightarrow X_r^p \dashrightarrow D_p^r \rightarrow Q_p$, and so $Q_r \rightarrow Q_p$. (Claim 5.54.3) □

Claim 5.54.4. *Process r is CS-qualified before any process in Y starts the exit protocol.*

Proof. By Claim 5.54.3 and the assumption that process p is CS-qualified before any process in Y starts the exit protocol, it follows that r is CS-qualified before any process in Y starts the exit protocol. (Claim 5.54.4) □

By Claim 5.54.1, r chooses a ticket t_r such that for all $i \in Y$, $(t_r, r) > (t_i, i)$, where t_i is the ticket chosen by i. By Claim 5.54.2, $\forall i \in Y, C_i \dashrightarrow A_r$, and by Claim 5.54.4, r is CS-qualified before any process in Y starts the exit protocol. Thus when r becomes CS-qualified (i.e., Q_r finishes) the statement of the lemma is violated. By Claim 5.54.3, $Q_r \rightarrow Q_p$, and so r's passage violates the statement of the lemma prior to the start of Q_p. This contradicts the assumption that the lemma is not violated prior to the start of Q_p. □

Using the preceding lemma, it is straightforward to prove k-exclusion.

Lemma 5.55. *The algorithm in Figure 5.10 satisfies k-exclusion.*

Proof. Suppose, by way of contradiction, that $k + 1$ processes are in the CS concurrently. Let Y be this set of processes, and let p be the process in Y with the largest $(ticket, process\_id)$ pair. All the processes in Y are in the CS concurrently, and so $\forall i \in Y \setminus \{p\}, C_i \rightarrow EP_p$. From this, the assumption that p has the largest $(ticket, process\_id)$ pair, and Corollary 5.49, it follows that $\forall i \in Y \setminus \{p\}, C_i \dashrightarrow A_p$. Therefore, by Lemma 5.54, process p is not CS-qualified until some process in Y leaves the CS, which means that p cannot be in the CS concurrently with all other processes in Y. This contradicts the assumption that p is in the CS concurrently with all other processes in Y. □

The following lemma is used to prove that FIFE holds, and that a process has $O(N)$ RMR complexity in the CC model. This lemma is the analogue of Lemma 5.27, but is significantly more complex to prove as the result of the presence of multi-reader multi-writer variables.

Lemma 5.56. *If a process p is non-faulty and finishes the doorway before a process q starts the doorway, then after q enters the CS, p enters the CS after taking $O(N)$ steps.*

Proof. Assume that a process p is non-faulty and finishes the doorway before a process q starts the doorway. We need to show that after q enters the CS, p enters the CS after

taking $O(N)$ steps. Suppose, for contradiction, that this is not the case. Let CS_p be the operation in which p executes the CS, and let CS_q be the operation in which q executes its first step in the CS. (If CS_q does not exist, i.e., q never enters the CS, then the lemma holds, so assume that CS_q exists. We have no guarantee at this point that p enters the CS, but we define the CS_p operation now and are careful to make sure it exists if necessary in our reasoning below.) Furthermore, let t_p and t_q be the tickets chosen by p and q, respectively, at line 173.

Let R^C_{p3} and R^{C+}_{p3} be p's first and second (in the sense of the total order given by $\rightarrow$) operations in the same passage by p that read $Capture[q][p][3]$ at line 191 such that $CS_q \dashrightarrow R^C_{p3}$. (These operations exist by inspection of the algorithm, the assumption that p is non-faulty, and the assumption that after q enters the CS, p does not enter the CS after taking $O(N)$ steps.)

Claim 5.56.1. $t_p < t_q$.

Proof. Since p finishes the doorway before q starts the doorway, $A_p \rightarrow C_q$. If p executes the exit protocol, then by the structure of the algorithm and $CS_q \dashrightarrow R^C_{p3}$, we have that $C_q \rightarrow CS_q \dashrightarrow R^C_{p3} \rightarrow EP_p$. Thus $A_p \rightarrow C_q$, and either $C_q \rightarrow EP_p$ or p never executes the exit protocol. Hence, by Lemma 5.48, $t_p < t_q$. □

Let R^T_q be the read by process q of $Ticket[p][q]$ at line 195. (This read exists by the structure of the algorithm and the fact that CS_q exists. Only one such read is performed at line 195, although the value read is used in three separate comparisons.) Let $W^T_p \subseteq A_p$ be p's write of t_p to $Ticket[p][q]$ at line 174. (This operation exists since p finishes the doorway by assumption).

Claim 5.56.2. *If $\hat{R}$ is R^T_q, or $\hat{R}$ is a read by q of $Ticket[p][q]$ at line 195 such that $R^T_q \rightarrow \hat{R} \rightarrow CS_p$, then $\hat{R}$ returns the same value as written by W^T_p, i.e. t_p.*

Proof. Assume that $\hat{R}$ is R^T_q, or that $\hat{R}$ is a read by q of $Ticket[p][q]$ at line 195 such that $R^T_q \rightarrow \hat{R} \rightarrow CS_p$. Also, let W^{T+}_p be p's first write to $Ticket[p][q]$ after W^T_p, i.e., in W^{T+}_p,

p writes 0 to $Ticket[p][q]$ at line 198. (Note that W_p^{T+} may not exist.)

The assumption that process p finishes the doorway before q starts the doorway, and the fact that W_p^T and C_q are part of the doorway, imply $W_p^T \to C_q \to R_q^T$, and so $W_p^T \to R_q^T$. Therefore $W_p^T \to \hat{R}$ (since $\hat{R}$ is R_q^T or $R_q^T \to \hat{R}$). Process p is the only process that writes $Ticket[p][q]$, and so if W_p^{T+} does not exist, then W_p^T is the uniquely defined last write to $Ticket[p][q]$ that finishes before $\hat{R}$ starts, and we're done. So assume that W_p^{T+} exists. We will now show that $\hat{R} \to W_p^{T+}$. Once we do so, since p is the only process to write $Ticket[p][q]$, and W_p^T and W_p^{T+} are successive writes by p, it follows that $\hat{R}$ must read the value written by W_p^T, i.e., t_p.

It remains to show that $\hat{R} \to W_p^{T+}$. Suppose, for contradiction, that $W_p^{T+} \dashrightarrow \hat{R}$. Since W_p^{T+} exists and is a write at line 198 in the exit protocol, it follows that CS_p exists. Furthermore, by the structure of the algorithm and the assumption that $\hat{R} \to CS_p$, we have that $CS_p \to W_p^{T+} \dashrightarrow \hat{R} \to CS_p$, which implies that $CS_p \to CS_p$. This contradicts that $\to$ is an irreflexive relation. Thus $\hat{R} \to W_p^{T+}$, as desired. □

Let R_{q1}^C, R_{q2}^C, R_{q3}^C be q's read of $Capture[q][p][1]$, $Capture[q][p][2]$, and $Capture[q][p][3]$ at line 195. (These reads exists by the structure of the algorithm and the fact that CS_q exists.) Let W_{p1}^C, W_{p2}^C, and W_{p3}^C be p's write of 0 to $Capture[q][p][1]$, $Capture[q][p][2]$, and $Capture[q][p][3]$ at line 176. (These writes exists since p finishes the doorway, by assumption.)

Either q evaluates the condition at line 195 to be true or not.

Case 1: q evaluates the condition at line 195 to be true.

In this case, let W_{q1}^C, W_{q2}^C, and W_{q3}^C, be q's write of t_q to $Capture[q][p][1]$, $Capture[q][p][2]$, and $Capture[q][p][3]$ at line 196. (These writes exist by the structure of the algorithm, the assumption of this case, and the assumption that CS_q exists).

Recall that R_{p3}^C and R_{p3}^{C+} are p's first and second read of $Capture[q][p][3]$ at line 191 such that $CS_q \dashrightarrow R_{p3}^C$.

Claim 5.56.3. *R_{p3}^{C+} reads t_q.*

Proof. By $CS_q \dashrightarrow R_{p3}^{C}$ and the structure of the algorithm, $W_{q3}^{C} \rightarrow CS_q \dashrightarrow R_{p3}^{C} \rightarrow R_{p3}^{C+}$, and so $W_{q3}^{C} \rightarrow R_{p3}^{C+}$. We will now show that R_{p3}^{C+} reads the same value as written by W_{q3}^{C}, i.e., t_q. To do this, we show that for any write $\hat{W}$ to *Capture*$[q][p][3]$ other than W_{q3}^{C}, either $\hat{W} \rightarrow W_{q3}^{C}$ or $R_{p3}^{C+} \rightarrow \hat{W}$.

Process p and q are the only processes that write *Capture*$[q][p][3]$.

Claim 5.56.3.Case 1: $\hat{W}$ is a write to *Capture*$[q][p][3]$ by process q.

For any write $\hat{W}$ to *Capture*$[q][p][3]$ by q other than W_{q3}^{C}, either $\hat{W} \rightarrow W_{q3}^{C}$ or $W_{q3}^{C} \rightarrow \hat{W}$. If $\hat{W} \rightarrow W_{q3}^{C}$, then we're done. So assume that $W_{q3}^{C} \rightarrow \hat{W}$. It suffices to show that $R_{p3}^{C+} \rightarrow \hat{W}$ for the first write $\hat{W}$ by q such that $W_{q3}^{C} \rightarrow \hat{W}$. Suppose, for contradiction that $\hat{W} \dashrightarrow R_{p3}^{C+}$.

Let $R_{q1}^{C'}$ be q's read of *Capture*$[q][p][1]$ at line 195 immediately before $\hat{W}$, and let $R_{q}^{T'}$ be q's read of *Ticket*$[p][q]$ at line 195 immediately before $\hat{W}$. (These operations exist by the structure of the algorithm.) The condition at line 195 must be true for $\hat{W}$ to exist; assume, without loss of generality, that $R_{q1}^{C'}$ returns a value that is $\leq$ the value returned by $R_{q}^{T'}$. (The argument that follows is similar if it is one of the other parts of the disjunction at line 195 that evaluates to true.)

By the assumption that $\hat{W} \dashrightarrow R_{p3}^{C+}$, and the structure of the algorithm, we have that $R_{q}^{T'} \rightarrow \hat{W} \dashrightarrow R_{p3}^{C+} \rightarrow CS_p$, and so $R_{q}^{T'} \rightarrow CS_p$. Furthermore, $R_{q}^{T} \rightarrow R_{q}^{T'}$, and so $R_{q}^{T} \rightarrow R_{q}^{T'} \rightarrow CS_p$. By Claim 5.56.2, $R_{q}^{T'}$ returns the same value as written by W_{p}^{T}, i.e., t_p. We will now prove that $R_{q1}^{C'}$ returns t_q. Once we do so, we can make the following argument. In the preceding paragraph we established that $R_{q1}^{C'}$ returns a value that is $\leq$ the value returned by $R_{q}^{T'}$, and so $t_q \leq t_p$, i.e. $t_p \geq t_q$. This contradicts Claim 5.56.1, which says that $t_p < t_q$. Thus, the supposition that $\hat{W} \dashrightarrow R_{p3}^{C+}$ is false, and $R_{p3}^{C+} \rightarrow \hat{W}$, as desired. It remains to prove that $R_{q1}^{C'}$ returns t_q.

$R_{q1}^{C'}$ occurs in later passage by q than W_{q1}^{C}, and so $W_{q1}^{C} \rightarrow R_{q1}^{C'}$. We will prove that $R_{q1}^{C'}$ returns the same value written as W_{q1}^{C}, i.e., t_q. To do this, we show that for any

write $\bar{W}$ to *Capture*[q][p][1] other than W^C_{q1}, either $\bar{W} \rightarrow W^C_{q1}$ or $R^{C'}_{q1} \rightarrow \bar{W}$.

Process p and q are the only processes that write *Capture*[q][p][1]. First we consider writes by process p. Process p finishes the doorway before q starts the doorway, and W^C_{p1} is part of the doorway, so $W^C_{p1} \rightarrow W^C_{q1}$. Furthermore, by the assumption that $\hat{W} \dashrightarrow R^{C+}_{p3}$ and the structure of the algorithm, $R^{C'}_{q1} \rightarrow \hat{W} \dashrightarrow R^{C+}_{p3} \rightarrow CS_p$, which implies that $R^{C'}_{q1} \rightarrow CS_p$. Thus $W^C_{p1} \rightarrow W^C_{q1} \rightarrow R^{C'}_{q1} \rightarrow CS_p$. Process p only writes *Capture*[p][q][1] once per passage, and so for any write $\bar{W}$ by p to *Capture*[q][p][1], either $\bar{W} \rightarrow W^C_{q1}$ (if $\bar{W}$ is W^C_{p1} or an earlier write) or $R^{C'}_{q1} \rightarrow CS_p \rightarrow \bar{W}$ (if $\bar{W}$ occurs in a passage later than W^C_{p1}). Thus, either $\bar{W} \rightarrow W^C_{q1}$ or $R^{C'}_{q1} \rightarrow \bar{W}$, as desired.

Next, consider writes $\bar{W}$ by process q. Recall that $\hat{W}$ is the first write by q to *Capture*[q][p][3] after W^C_{q3}, and $R^{C'}_{q1}$ is the read of *Capture*[q][p][1] that immediately precedes $\hat{W}$. Process q writes *Capture*[q][p][1] and *Capture*[q][p][3] at most once per passage (at line 196), after the condition at line 195 is evaluated. So, by the structure of the algorithm, the first write by q to *Capture*[q][p][1] after W^C_{q1} finishes can start only after $R^{C'}_{q1}$ finishes. Therefore, for any write $\bar{W}$ by q to *Capture*[q][p][1] other than W^C_{q1}, either $\bar{W} \rightarrow W^C_{q1}$ (if $\bar{W}$ occurs in a passage before the one in which W^C_{q1} and W^C_{q3} occur), or $R^{C'}_{q1} \rightarrow \bar{W}$ (if $\bar{W}$ occurs in the same or later passage as $\hat{W}$), as desired.

Claim 5.56.3.Case 2: $\hat{W}$ is a write to *Capture*[q][p][3] by process p.

In this case we want to show that for any write $\hat{W}$ to *Capture*[q][p][3] by p, either $\hat{W} \rightarrow W^C_{q3}$ or $R^{C+}_{p3} \rightarrow \hat{W}$.

Process p only writes *Capture*[q][p][3] once per passage, at line 176. Therefore, any write by p to *Capture*[q][p][3] after W^C_{p3} must occur in a later passage than the one in which W^C_{p3} occurs. Since W^C_{p3} and R^{C+}_{p3} occur in the same passage, any write by p to *Capture*[q][p][3] after W^C_{p3} must occur in a later passage than the one in which R^{C+}_{p3} occurs. Furthermore, process p finishes the doorway before q starts the doorway, W^C_{p3} is part of the doorway, and so $W^C_{p3} \rightarrow W^C_{q3}$. Thus, for any write $\hat{W}$ to *Capture*[q][p][3] by p, either $\hat{W} \rightarrow W^C_{q3}$ (if $\hat{W}$ is W^C_{p3} or an earlier write) or $R^{C+}_{p3} \rightarrow \hat{W}$ (if $\hat{W}$ occurs in a passage

later than the one in which W_{p3}^{C} and R_{p3}^{C+} occur), as desired. □

By Claim 5.56.3, R_{p3}^{C+} reads t_q, and by Claim 5.56.1, $t_p < t_q$. Thus p evaluates the condition at line 191 to be true. An argument similar to the one in Claim 5.56.3 can be used to establish that the value returned by p's next read of $Capture[q][p][2]$ at line 192 is t_q. Using this, we can establish that the condition p evaluates at line 192 is true. Finally, an analogous argument can be made that p subsequently evaluates the condition at line 193 to be true. After this, p sets *captured* to be true, and, by inspection of the algorithm, executes $O(N)$ more steps prior to entering the CS.

We have established the following: After q enters the CS, p executes $O(N)$ steps before starting R_{p3}^{C+} at line 191. This follows by the structure of the algorithm and because R_{p3}^{C} and R_{p3}^{C+} are p's first and second reads of $Capture[q][p][3]$ at line 191 such that $CS_q \dashrightarrow R_{p3}^{C}$. After finishing R_{p3}^{C+}, p executes $O(N)$ more steps prior to entering the CS. Therefore, after q enters the CS, p executes $O(N)$ steps in the trying protocol before entering the CS. This contradicts that after q enters the CS, p does not enter the CS after taking $O(N)$ steps.

Case 2: q evaluates the condition at line 195 to be false.

By Claim 5.56.2, R_q^T returns t_p. By assumption of this case, q evaluates the condition at line 195 to be false, and so R_{q3}^{C}, R_{q2}^{C}, and R_{q1}^{C} each return a value strictly greater than t_p.

Recall that R_{p3}^{C} and R_{p3}^{C+} are p's first and second read of $Capture[q][p][3]$ at line 191 such that $CS_q \dashrightarrow R_{p3}^{C}$.

Claim 5.56.4. *R_{p3}^{C+} reads the same value as R_{q3}^{C}.*

Proof. By the structure of the algorithm and the assumption that $CS_q \dashrightarrow R_{p3}^{C}$, $R_{q3}^{C} \rightarrow CS_q \dashrightarrow R_{p3}^{C} \rightarrow R_{p3}^{C+}$, and so $R_{q3}^{C} \rightarrow R_{p3}^{C+}$. We now prove that for every write $\hat{W}$ to $Capture[q][p][3]$, either $\hat{W} \rightarrow R_{q3}^{C}$, or $R_{p3}^{C+} \rightarrow \hat{W}$. This establishes that R_{q3}^{C} and R_{p3}^{C+} read the same value.

Process p and q are the only processes that write *Capture*$[q][p][3]$.

Claim 5.56.4.Case 1: $\hat{W}$ is a write to *Capture*$[q][p][3]$ by process q.

By assumption of this case, q does not write *Capture*$[q][p][3]$ in the same passage as R^C_{q3} occurs. Thus, for any write $\hat{W}$ to *Capture*$[q][p][3]$ by q, either $\hat{W} \rightarrow R^C_{q3}$ (if $\hat{W}$ occurs in a passage earlier than the one in which R^C_{q3} occurs) or $R^C_{q3} \rightarrow \hat{W}$ (if $\hat{W}$ occurs in a passage later than the one in which R^C_{q3} occurs). If $\hat{W} \rightarrow R^C_{q3}$, then we are done. So assume that $R^C_{q3} \rightarrow \hat{W}$. It suffices to show that $R^{C+}_{p3} \rightarrow \hat{W}$ for the first write $\hat{W}$ to *Capture*$[q][p][3]$ by q such that $R^C_{q3} \rightarrow \hat{W}$. Suppose, for contradiction that $\hat{W} \dashrightarrow R^{C+}_{p3}$.

Let $R^{C'}_{q1}$ be q's read of *Capture*$[q][p][1]$ at line 195 immediately before $\hat{W}$, and let $R^{T'}_q$ be q's read of *Ticket*$[p][q]$ at line 195 immediately before $\hat{W}$. (These operations exist by the structure of the algorithm.) The condition at line 195 must be true for $\hat{W}$ to exist; assume, without loss of generality, that $R^{C'}_{q1}$ returns a value that is $\leq$ the value returned by $R^{T'}_q$. (The argument that follows is similar if it is one of the other parts of the disjunction at line 195 that evaluates to true.)

By the assumption that $\hat{W} \dashrightarrow R^{C+}_{p3}$, and the structure of the algorithm, we have that $R^{T'}_q \rightarrow \hat{W} \dashrightarrow R^{C+}_{p3} \rightarrow CS_p$, and so $R^{T'}_q \rightarrow CS_p$. This, and the fact that $R^{T'}_q$ occurs in a later passage than R^T_q, imply $R^T_q \rightarrow R^{T'}_q \rightarrow CS_p$. By Claim 5.56.2, $R^{T'}_q$ returns the same value as written by W^T_p, i.e., t_p. In the preceding paragraph we established that $R^{C'}_{q1}$ returns a value v that is $\leq$ the value returned by $R^{T'}_q$, and so $v \leq t_p$. We will now prove that $R^{C'}_{q1}$ returns the same value v as R^C_{q1}. Once we do so, we can make the following argument. Recall that by assumption of this case, the value returned by R^C_{q1} is strictly greater than t_p. Therefore $v > t_p$, which contradicts that $v \leq t_p$. Thus, the supposition that $\hat{W} \dashrightarrow R^{C+}_{p3}$ is false, and $R^{C+}_{p3} \rightarrow \hat{W}$, as desired. It remains to prove that $R^{C'}_{q1}$ returns the same value as R^C_{q1}.

$R^{C'}_{q1}$ occurs in a passage after R^C_{q1}, so $R^C_{q1} \rightarrow R^{C'}_{q1}$. To show that $R^{C'}_{q1}$ returns the same value as R^C_{q1}, we show that for any write $\bar{W}$ to *Capture*$[q][p][1]$, $\bar{W} \rightarrow R^C_{q1}$ or $R^{C'}_{q1} \rightarrow \bar{W}$.

Process p and q are the only processes that write *Capture*$[q][p][1]$.

First we consider writes $\bar{W}$ by process p. W_{p1}^C is part of p's doorway, and p finishes the doorway before q starts the doorway, so $W_{p1}^C \rightarrow R_{q1}^C$. Furthermore, by assumption, $\hat{W} \dashrightarrow R_{p3}^{C+}$, and so $R_{q1}^{C'} \rightarrow \hat{W} \dashrightarrow R_{p3}^{C+} \rightarrow CS_p$, which implies that $R_{q1}^{C'} \rightarrow CS_p$. Thus $W_{p1}^C \rightarrow R_{q1}^C \rightarrow R_{q1}^{C'} \rightarrow CS_p$. Process p only writes $Capture[p][q][1]$ once per passage, and so for any write $\bar{W}$ by p to $Capture[q][p][1]$, either $\bar{W} \rightarrow R_{q1}^C$ (if $\bar{W}$ is W_{p1}^C or an earlier write) or $R_{q1}^{C'} \rightarrow CS_p \rightarrow \bar{W}$ (if $\bar{W}$ occurs in a passage later than the one in which W_{p1}^C and CS_p occur). Thus, either $\bar{W} \rightarrow R_{q1}^C$ or $R_{q1}^{C'} \rightarrow \bar{W}$, as desired.

Next, consider writes $\bar{W}$ by process q. Recall that $\hat{W}$ is the first write by q to $Capture[q][p][3]$ after R_{q3}^C, and $R_{q1}^{C'}$ is the read of $Capture[q][p][1]$ that immediately precedes $\hat{W}$. Process q writes $Capture[q][p][1]$ and $Capture[q][p][3]$ at most once per passage (at line 196), after the condition at line 195 is evaluated, and no writes by q to $Capture[q][p][3]$ or $Capture[q][p][1]$ occur in the same passage that R_{q1}^C and R_{q3}^C occur. Therefore, for any write $\bar{W}$ by q to $Capture[q][p][1]$, either $\bar{W} \rightarrow R_{q1}^C$ (if $\bar{W}$ occurs in a passage before the one in which R_{q1}^C and R_{q3}^C occur), or $R_{q1}^{C'} \rightarrow \bar{W}$ (if $\bar{W}$ occurs in the same or later passage as $\hat{W}$), as desired.

Claim 5.56.4.Case 2: $\hat{W}$ is a write to $Capture[q][p][3]$ by process p.

In this case we want to show that for any write $\hat{W}$ to $Capture[q][p][3]$ by p, either $\hat{W} \rightarrow R_{q3}^C$ or $R_{p3}^{C+} \rightarrow \hat{W}$.

Process p only writes $Capture[q][p][3]$ once per passage, at line 176. Therefore, any write by p to $Capture[q][p][3]$ after W_{p3}^C must occur in a later passage than the one in which W_{p3}^C occurs. Since W_{p3}^C and R_{p3}^{C+} occur in the same passage, any write by p to $Capture[q][p][3]$ after W_{p3}^C must occur in a later passage than the one in which R_{p3}^{C+} occurs. Furthermore, process p finishes the doorway before q starts the doorway, W_{p3}^C is part of the doorway, and so $W_{p3}^C \rightarrow R_{q3}^C$. Thus, for any write $\hat{W}$ to $Capture[q][p][3]$ by p, either $\hat{W} \rightarrow R_{q3}^C$ (if $\hat{W}$ is W_{p3}^C or an earlier write) or $R_{p3}^{C+} \rightarrow \hat{W}$ (if $\hat{W}$ occurs in a passage later than the one in which W_{p3}^C and R_{p3}^{C+} occur), as desired. □

By Claim 5.56.4, R_{p3}^{C+} returns the same value as R_{q3}^C. Immediately before Claim 5.56.4,

we established that R^C_{q3} returns a value strictly greater than t_p. Thus p evaluates the condition at line 191 to be true. An argument similar to the one in Claim 5.56.4 can be used to establish that the value returned by p's next read of $Capture[q][p][2]$ at line 192 is the same as the value returned by R^C_{q2}. Using this, we can establish that the condition p evaluates at line 192 is true. Finally, an analogous argument can be made that p subsequently evaluates the condition at line 193 to be true. After this, p sets *captured* to be true, and, by inspection of the algorithm, executes $O(N)$ more steps prior to entering the CS.

We have established the following: After q enters the CS, p executes $O(N)$ steps before starting R^{C+}_{p3} at line 191. This follows by the structure of the algorithm and because R^C_{p3} and R^{C+}_{p3} are p's first and second reads of $Capture[q][p][3]$ at line 191 such that $CS_q \dashrightarrow R^C_{p3}$. After finishing R^{C+}_{p3}, p executes $O(N)$ more steps prior to entering the CS. Therefore, after q enters the CS, p executes $O(N)$ steps in the trying protocol before entering the CS. This contradicts that after q enters the CS, p does not enter the CS after taking $O(N)$ steps. □

Lemma 5.57. *The algorithm in Figure 5.10 satisfies FIFE.*

Proof. Follows immediately from Lemma 5.56. □

Lemma 5.58. *In the CC model, a process p makes $O(N)$ RMRs in the loop at line 186.*

Proof. Either process p makes at most $60N - 1$ RMRs in the loop at line 186, or at least $60N$ RMRs. In the former case, the lemma is clearly true. In the latter case, let $R_{\leq 60}$ denote the operation in which p makes its first $60N$ RMRs. There are at most $6N$ shared variables ($Bypass[i][p]$, $Doorway[i][p]$, $Ticket[i][p]$, $Capture[i][p][1]$, $Capture[i][p][2]$, and $Capture[i][p][3]$ for each $i \in \{1..N\}$) that p reads in the loop, and so p makes at least ten RMRs to one of these variables before $R_{\leq 60}$ finishes. We consider the possible variables in six separate cases:

Case 1: Process p makes at least ten RMRs to $Bypass[q][p]$ for some process q. Let $R_{\leq 10} \subseteq R_{\leq 60}$ be the operation in which p makes these RMRs, let $R_p^i \subseteq R_{\leq 10}$ for $i \in \{1..10\}$ be the operation in which p makes its i'th RMR in $R_{\leq 10}$. Recall that in the CC model a process makes an RMR when it reads a variable for the first time, writes a variable, and whenever a process reads a variable for the first time after another process has written that variable. Process p only writes $Bypass[q][p]$ at line 183, outside of the loop at line 186. So, during $R_{\leq 10}$ (i.e., after $R_{\leq 10}$ starts and before R_{10} finishes), q must update $Bypass[q][p]$ at least nine times: p's first RMR in $R_{\leq 10}$ may happen because p does not have copy of $Bypass[q][p]$ in its cache, but each of p's nine subsequent RMRs to $Bypass[q][p]$ in $R_{\leq 10}$ can only occur after q writes $Bypass[q][p]$, invalidating the copy of $Bypass[q][p]$ in p's cache. More precisely, there must exist distinct write operations W_q^j by q to $Bypass[q][p]$, for $j \in \{1..9\}$, such that $R_p^j \dashrightarrow W_q^j \rightarrow R_p^{j+1}$. This and the fact that process q writes $Bypass[q][p]$ at most once per passage (line 200) imply that there is a passage by q that occurs during $R_{\leq 10}$. More precisely, there is a passage P_q by process q such that $R_p^1 \rightarrow P_q$ and $P_q \rightarrow R_p^{10}$. Let the operation DWY_q denote q's execution of its doorway in P_q, and let DWY_p denote p's execution of its doorway. $DWY_p \rightarrow R_p^1 \rightarrow P_q$, and so $DWY_p \rightarrow R_p^1 \rightarrow DWY_q$. This implies that p finishes its doorway before q starts its doorway. By this and Lemma 5.56, after q enters the CS, p enters the CS after taking $O(N)$ steps. Process q's passage P_q finishes before R_p^{10} starts (since $P_q \rightarrow R_p^{10}$), and so q enters the CS before p finishes $R_{\leq 10}$. Therefore, after $R_{\leq 10}$ finishes, p enters the CS after taking $O(N)$ steps. This implies that p makes $O(N)$ additional RMRs in the loop at line 186 after $R_{\leq 10}$ finishes. **(End of Case 1)**

The argument for the remaining cases is similar: it involves showing that there exists a passage by q that occurs entirely during $R_{\leq 10}$, and then applying Lemma 5.56 to show that p makes at most $O(N)$ additional RMRs in the loop at line 186 after $R_{\leq 10}$ finishes. We summarize the cases below, but the details are omitted.

Case 2: Process p makes at least ten RMRs to $Doorway[q][p]$ for some process q.

The argument for this case is similar to Case 1, except that q writes $Doorway[q][p]$ twice per passage (at line 171 and line 182).

Case 3: Process p makes at least ten RMRs to $Ticket[q][p]$ for some process q. The argument for this case is similar to Case 1, except that q writes $Ticket[q][p]$ twice in a passage (at line 174 and line 198).

Case 4-6: Process p makes at least ten RMRs to $Capture[q][p][*]$ for some process q. The argument for this case is similar to Case 1, except that q writes $Capture[q][p][*]$ at most once per passage (at line 196). □

Lemma 5.59. *The algorithm in Figure 5.10 has RMR complexity $\Theta(N)$ in both the DSM and CC models.*

Proof. We first prove the lemma for the CC model. At each line in the doorway and the exit protocol, a process p makes $\Theta(N)$ RMRs, and so p makes a total of $\Theta(N)$ RMRs in the doorway and exit protocol. In the waiting room, outside of the loop at line 186, p makes $\Theta(N)$ RMRs. By Lemma 5.58, p makes $O(N)$ RMRs in the loop at line 186. Therefore, the total number of RMRs p makes in a passage is $\Theta(N)$.

The result also follows for the DSM model by a similar argument. The only difference in the DSM model is that p makes no RMRs while in the loop at line 186, as each shared variable referenced in that loop is local to p. □

Theorem 5.60. *The algorithm in Figure 5.10 satisfies k-exclusion, starvation freedom, and FIFE. Moreover, it has RMR complexity $\Theta(N)$ in both the DSM and CC models.*

Proof. The result follows from Lemmas 5.55, 5.41 (modulo line number changes), 5.57, and 5.59. □

5.11 Summary of Techniques

This chapter introduced a number of different k-exclusion algorithms. All of them satisfy k-exclusion, starvation freedom, bounded exit, and k-FCFS. Additionally, all of them

have $O(N)$ RMR complexity in both the CC and DSM models, except for the algorithm presented in Section 5.5, which has $O(N)$ RMR complexity only in the CC model. The algorithms differ in the set of features identified in Section 5.2: ticket resetting, strength of read and write operations required (atomic vs. non-atomic), and FIFE. A number of different key techniques are used by the algorithms to satisfy these features. We summarize these techniques here.

The simplest algorithm, which does not reset tickets, requires atomic reads and writes, and does not satisfy FIFE, is presented in Section 5.4. To satisfy FIFE, the algorithm is augmented in Section 5.6 with a *capturing* mechanism: processes, prior to entering the CS, capture other processes with smaller tickets that are still in the waiting room, thereby granting them entry into the CS. To modify this algorithm to use only non-atomic reads and writes, we used in Section 5.7 the idea of *shared variable duplication*: duplicate certain shared variables, and then read the copies of those variables in the reverse order of which they are written.

The first ticket resetting algorithm that we introduced was in Section 5.8. The key technique in this algorithm is the *detection of waiting-room bypass*: if a process p is in the waiting room, and another process q executes a full passage, thereby "bypassing" p in the waiting room, then p detects this and removes q from $p.predecessor\_set$. This algorithm works with non-atomic reads and writes, even though it does not use the shared variable duplication technique as in Section 5.7. In Section 5.9, the k-exclusion algorithm we present uses atomic reads and writes, resets tickets, and additionally satisfies FIFE. To satisfy FIFE, the algorithm uses a capturing mechanism, similar to the one used earlier in Section 5.6. However, simply using a capturing mechanism is not sufficient to arrive at a correct k-exclusion algorithm. The algorithm uses another technique, in conjunction with capturing, that we call *detection of doorway bypass*: if a process p is in the doorway, and a process q executes through the doorway, thereby "bypassing" p in the doorway, then p should be able to detect this. This allows p to apply the capturing mechanism

only in situations that will not lead to k-exclusion being violated.

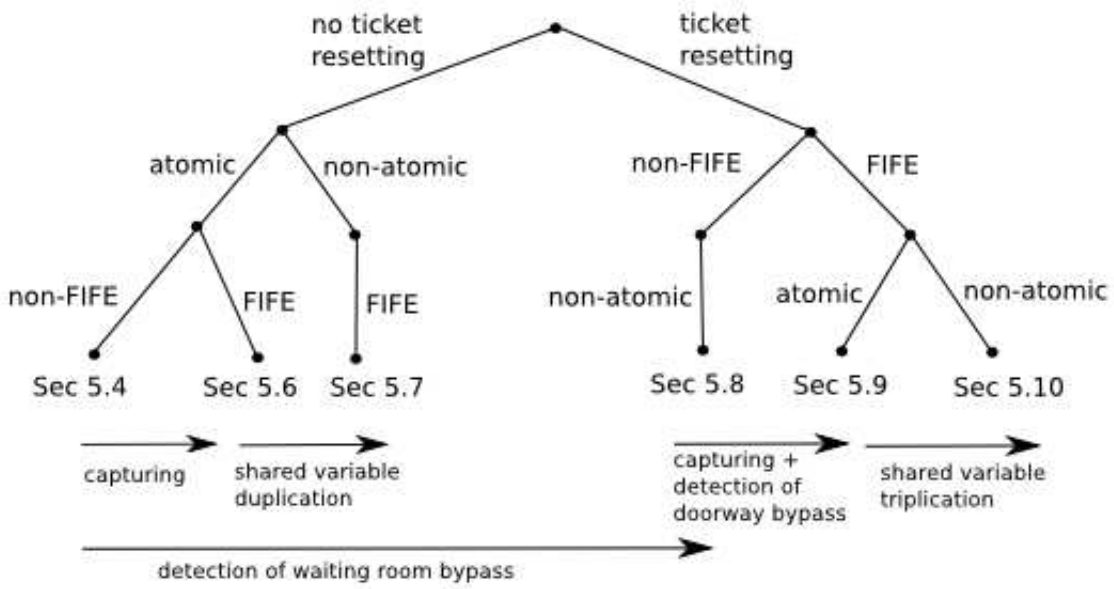

Figure 5.11: Summary of k-exclusion algorithms, features, and techniques.

To arrive at the algorithm in Section 5.10, which resets tickets, requires only non-atomic reads and writes, and satisfies FIFE, we modify the algorithm from Section 5.9 by using *shared variable triplication*: triplicate certain shared variables, and then read the copies of those variables in the reverse order of which they are written.

The tree diagram in Figure 5.11 summarizes the relationship between the different algorithms, the features that they possess, and the key techniques that they use.

Chapter 6

Conclusion

We have presented in this thesis a number of novel local-spin algorithms for variants of the mutual exclusion problem, using only read and write operations on shared variables as the synchronization primitives. In particular, we have presented an efficient FCFS mutual exclusion algorithm that uses only atomic reads and writes and has $O(\log N)$ RMR complexity in both the DSM and CC models. To our knowledge, this is the first such algorithm presented in the literature. Prior FCFS mutual exclusion algorithms either have super-logarithmic RMR complexity, or use synchronization primitives stronger than atomic reads and writes. This algorithm is also adaptive to point contention. More precisely, the number of RMRs a process makes per passage is $\Theta(\min(c, \log N))$, where c is the point contention. We can use Lamport's register constructions [33, 34] to transform this algorithm into one that uses only non-atomic reads and writes, however this transformation is extremely inefficient and the result would not have $O(\log N)$ RMR complexity. It remains an open problem as to whether FCFS mutual exclusion can be done using non-atomic reads and writes and have $O(\log N)$ RMR complexity.

We have also presented a transformation that converts abortable mutual exclusion to FCFS abortable mutual exclusion. This transformation uses only reads and writes and is local-spin. In conjunction with work by Danek and Lee [15, 35] and work by Danek

and Hadzilacos [14], this transformation yields the first known FCFS abortable mutual exclusion algorithm and the first known group mutual exclusion algorithm that are local-spin and use only atomic reads and writes. Both algorithms have $O(N)$ RMR complexity in the DSM and CC models. It is not known whether FCFS abortable mutual exclusion can be done with $O(\log N)$ RMR complexity. There is no obvious way to adapt the $O(\log N)$ RMR complexity FCFS mutual exclusion algorithm from this thesis to make it abortable, as it uses a priority queue that is protected by an auxiliary lock. Any abort protocol that updates the priority queue would have to acquire the lock first, which means that bounded abort would not be satisfied.

Finally, we have presented the first known k-exclusion algorithms that are local-spin in the CC and DSM models and that use only atomic reads and writes. The only previously known local-spin k-exclusion algorithms are by Anderson and Moir [2]. Their algorithms have RMR complexity $\Theta(k \log (N/k))$ and $\Theta(c)$, where c is *point contention*, i.e., the maximum number of processes simultaneously outside of the NCS during a passage. Unlike our k-exclusion algorithms, Anderson and Moir's algorithms use strong synchronization primitives such as Fetch&Add, Test&Set, and Compare&Swap in addition to atomic reads and writes. Interestingly, the worst-case RMR complexity of their algorithms matches asymptotically the worst-case RMR complexity of the algorithms in this thesis, for any k that is a constant fraction of N (e.g., for $k = N/2$). This is significant because it leads to a state of affairs in which the only known local-spin k-exclusion algorithms that use strong synchronization primitives have the same worst-case RMR complexity (within constant factors) as the only known local-spin k-exclusion algorithms that use only atomic reads and writes. This is in contrast to what is known about the case in which $k = 1$ (i.e., ordinary mutual exclusion): Using stronger synchronization primitives like Fetch&Add there exist $O(1)$ RMR complexity mutual exclusion algorithms [4] (and references cited therein), whereas the class of mutual exclusion algorithms that use only atomic reads and writes (and comparison primitives) have $\Omega(\log N)$ RMR

complexity [5].

In light of the preceding paragraph, our results in Chapter 5 open some interesting questions. For a given set of synchronization primitives, what exactly is the RMR complexity of k-exclusion as k varies? Is the RMR complexity of k-exclusion (asymptotically) the same as that of mutual exclusion for all values of k, or are there values of k for which k-exclusion is (asymptotically) harder in terms of RMRs than mutual exclusion? This question can be asked with respect to algorithms that use only reads and writes, as well as algorithms that also use stronger synchronization primitives such as Fetch&Add. In the case of mutual exclusion it is known that the choice of primitives affects the RMR complexity. As pointed out above, however, this not (yet) known in the case of k-exclusion. We conjecture that there is a super-logarithmic lower bound for k-exclusion that holds irrespective of synchronization primitives for values of k that are a constant fraction of N (e.g., for $k = N/2$).

We also presented several variants of our k-exclusion algorithms that work even if read and write operations are not atomic. These algorithms are structured with the same elegance of Lamport's Bakery algorithm. Like the Bakery algorithm, however, tickets can grow without bound in our algorithms. There exist variations of Lamport's Bakery algorithm for mutual exclusion [27, 42] in which tickets are bounded and each shared variable requires only $\Theta(\log N)$ bits. The techniques used in these papers, unfortunately, are not easily adapted to our algorithms, since they rely on the fact that at most one process can execute through the CS at a time. Afek et al. [1] use a more generic scheme, known as a bounded concurrent timestamp system (BCTS) [17], to bound tickets in their Bakery-like (non-local-spin) k-exclusion algorithm. However, the BCTS implementation in [17] requires large shared variables, each with size $\Omega(N)$ bits. In fact, all bounded timestamp systems require $\Omega(N)$ bits per timestamp [24], and thus tend to be impractical. Furthermore, a process executing the algorithm of Afek et al. can make an unbounded number of calls to the BCTS while waiting to enter the CS, and it is not clear how these

calls can be made so as to incur only a bounded number of RMRs. Finding a variant of our algorithms that are local-spin and use bounded tickets, preferably requiring $O(\log N)$ bits per timestamp, remains an open problem.

Bibliography

[1] Yehuda Afek, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Trans. Program. Lang. Syst.*, 16(3):939–953, 1994.

[2] J. H. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11(1):1–20, 1997.

[3] James H. Anderson and Yong-Jik Kim. Nonatomic mutual exclusion with local spinning. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 3–12, New York, NY, USA, 2002. ACM.

[4] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-Memory Mutual Exclusion: Major Research Trends Since 1986. *Distributed Computing*, 2002.

[5] Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 217–226, New York, NY, USA, 2008. ACM.

[6] Vibhor Bhatt and Chien-Chung Huang. Group mutual exclusion in $O(\log N)$ RMR. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 45–54, New York, NY, USA, 2010. ACM.

[7] Vibhor Bhatt and Prasad Jayanti. On the existence of weakest failure detectors for mutual exclusion and k-exclusion. In *DISC'09: Proceedings of the 23rd international conference on Distributed computing*, pages 311–325, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 145–157, New York, NY, USA, 1989. ACM.

[9] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.

[10] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[11] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[12] Robert Danek. Local-spin group mutual exclusion algorithms. Master's thesis, University of Toronto, 2002.

[13] Robert Danek and Wojciech Golab. Closing the complexity gap between fcfs mutual exclusion and mutual exclusion. *Distributed Computing*, pages 1–25, 2010. 10.1007/s00446-010-0096-2.

[14] Robert Danek and Vassos Hadzilacos. Local-Spin Group Mutual Exclusion Algorithms. In *Proceedings of the 18th International Symposium on Distributed Computing*, pages 71–85, October 2004.

[15] Robert Danek and Hyonho Lee. Brief announcement: Local-spin algorithms for abortable mutual exclusion and related problems. In *DISC*, volume 5218 of *Lecture Notes in Computer Science*, pages 512–513. Springer, 2008.

[16] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, September 1965.

[17] D. Dolev and N. Shavit. Bounded concurrrent time-stamp systems are constructible. In *Proc. of 21st STOC*, pages 454–466, New York, NY, USA, 1989.

[18] Danny Dolev, Eli Gafni, and Nir Shavit. Toward a non-atomic era: ℓ-exclusion as a test case. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 78–92, New York, NY, USA, 1988. ACM.

[19] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure (preliminary report). In *IEEE Symposium on Foundations of Computer Science*, pages 234–254, 1979.

[20] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Distributed fifo allocation of identical resources using small shared space. *ACM Trans. Program. Lang. Syst.*, 11(1):90–114, 1989.

[21] Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 3–12, New York, NY, USA, 2007. ACM.

[22] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 5(2):164–189, 1983.

[23] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[24] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

[25] Prasad Jayanti. f-arrays: Implementation and Applications. In *Proceedings of the 22th Annual ACM Symposium on Principles of Distributed Computing*, July 2002.

[26] Prasad Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 295–304, New York, NY, USA, 2003. ACM.

[27] Prasad Jayanti, King Tan, Gregory Friedland, and Amir Katz. Bounding lamport's bakery algorithm. In *SOFSEM '01: Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics Piestany*, pages 261–270, London, UK, 2001. Springer-Verlag.

[28] Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

[29] Patrick Keane and Mark Moir. A simple local-spin group mutual exclusion algorithm. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 23–32, New York, NY, USA, 1999. ACM.

[30] Y.-J. Kim and J. Anderson. Adaptive mutual exclusion with local spinning. *Dist. Computing*, 19(3):197–236, 2007.

[31] Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[32] Leslie Lamport. A new Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[33] Leslie Lamport. On interprocess communication. Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[34] Leslie Lamport. On interprocess communication. Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[35] Hyonho Lee. *Thesis (forthcoming)*. PhD thesis, University of Toronto, 2010.

[36] Edward A. Lycklama and Vassos Hadzilacos. A first-come-first-served mutual exclusion algorithm with small communication variables. *ACM Transactions on Programming Language Systems*, 13(4):558–576, 1991.

[37] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[38] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[39] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[40] Michael L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 31–40, New York, NY, USA, 2002. ACM.

[41] Michael L. Scott and William N. Scherer. Scalable queue-based spin locks with timeout. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 44–52, New York, NY, USA, 2001. ACM.

[42] G. Taubenfeld. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. In *DISC*, pages 56–70, 2004.

[43] Jae-Heon Yang and James H. Anderson. A Fast, Scalable Mutual Exclusion Algorithm. *Distributed Computing*, 9(1):51–60, August 1995.